Implementation und Generation von Datenbankmasken in Java

Studienarbeit,
Eric Schellhammer

16. Dezember 1999

Inhaltsverzeichnis

I Benutzerhandbücher 9

1 Die gemeinsamen Grundelemente der Benutzerschnittstellen 13

1.1 Das Loginfenster . . . 13

1.2 Die Programmzustände . . . 14

1.3 Das Maskenfenster . . . 15

1.3.1 Das Hauptpanel . . . 16

1.3.2 Die Statuszeilen . . . 16

1.3.3 Die Knopfleiste . . . 17

1.3.4 Die Menüzeile . . . 18

1.4 Der Listenassistent . . . 20

2 *Flit* Benutzerhandbuch 23

2.1 Übereinstimmung mit den Grundfunktionen . . . 23

2.2 Beschreibung der Maskenelemente . . . 23

2.3 Schlagwörter . . . 25

2.4 umfassende Dokumente . . . 26

3 *MetaMask* Benutzerhandbuch 27

3.1 Das Loginfenster . . . 27

3.1.1 Laden aus der Datenbank . . . 28

3.1.2 Lesen von vorhandenen Masken . . . 29

3.2 Die Programmzustände . . . 29

3.3 Das *MetaMask*-Hauptfenster . . . 29

3.3.1 Die Knopfleiste . . . 30

3.4 Die Testmaske . . . 31

3.5 Beschreibungen der einzelnen Felder des Hauptpanels 32
3.6 Das Umpositionieren von Maskenelementen 34
3.7 Das Editieren von Auswahllisten 35

4 *Litera* Benutzerhandbuch 37
4.1 Beschreibung der Maskenelemente 37
4.2 Bestellungen 38
4.3 Einzelbestellungen und der Warenkorb 39

II Implementationsdetails 41

5 Die Programmstruktur 43
5.1 Überblick über die Struktur 43
5.2 Grundklassen: shared 47
5.2.1 Verzeichnis `dbis/mask/shared/ui/` 47
5.2.2 Verzeichnis `dbis/mask/shared/net/` 48
5.2.3 Verzeichnis `dbis/mask/shared/dok/` 49
5.3 Grundklassen: templates 50
5.3.1 Verzeichnis `dbis/mask/templates/ui/` 50
5.3.2 Verzeichnis `dbis/mask/templates/net/` 50
5.3.3 Verzeichnis `dbis/mask/templates/dok/` 50
5.4 die automatische erste Instantiierung 51
5.4.1 Klasse DokumentDatensatz 52
5.4.2 Klasse Kriterien 53
5.4.3 Klasse SQLAssistant 53
5.4.4 Klasse Arbeitsablauf 54
5.4.5 ResourceBundle `DB.properties` 55
5.4.6 ResourceBundle `Sprache.properties` 55
5.5 Die empfohlene Rollen-Verwaltung in Oracle 57
5.5.1 Schreibrechte 57
5.5.2 Die Rollen 57

6 Wie werden die Masken erweitert? 59

6.1 Daten, die *MetaMask* nicht automatisch instantiiert 59

6.1.1 Listenformate 59

6.1.2 das *OrderBy*-Menü 60

6.1.3 Werte der Datenbank-Anbindung 61

6.2 Ergänzungen des Hauptprogramms 61

6.2.1 Der Arbeitsablauf 61

6.2.2 Hinzufügen von Knöpfen im Hauptpanel 61

6.2.3 Hinzufügen von neuen Menüeinträgen 62

6.2.4 Hinzufügen von Textfeldern 63

6.2.5 Fehlermeldungen 64

7 Dokumentation der einzelnen Klassen 65

7.1 Klasse AblaufVerwalter 65

7.2 Klasse Arbeitsablauf 67

7.3 Klasse BackgroundReader 70

7.4 Interface CheckMenuListener 71

7.5 Interface CommandListener 71

7.6 Klasse ConnectionManager 71

7.7 Klasse DokumentDatensatz 72

7.8 Klasse ExpertSQLEditor 75

7.9 Klasse ExprErfdatumParser 75

7.10 Klasse ExprIntParser 76

7.11 Klasse ExprParser 77

7.12 Klasse ExprStringParser 79

7.13 Klasse FieldChecker 79

7.14 Klasse FieldCheckerFloat 80

7.15 Klasse FieldCheckerFloatNE 80

7.16 Klasse FieldCheckerInt 81

7.17 Klasse FieldCheckerIntNE 81

7.18 Klasse FieldCheckerString 82

7.19 Klasse FieldCheckerStringNE 82

7.20 Klasse FieldCheckerStringNEQ 83

7.21 Klasse FieldCheckerStringNQ 83

7.22 Klasse FieldCheckerYear 84
7.23 Klasse Flit 84
7.24 Klasse FlitApplet 85
7.25 Klasse FlitButton 85
7.26 Klasse FlitButtonBar 86
7.27 Klasse FlitErrorDialog 86
7.28 Interface FlitFocusListener 87
7.29 Klasse FlitFrame 88
7.30 Klasse FlitInfoFrame 91
7.31 Klasse FlitKeyListener 92
7.32 Klasse FlitListFrame 93
7.33 Klasse FlitLoginFrame 95
7.34 Klasse FlitSpeedBar 96
7.35 Klasse FlitStrich 96
7.36 Klasse FlitTextArea 98
7.37 Klasse FlitTextField 98
7.38 Klasse FlitTextPanel 98
7.39 Klasse Kriterien 99
7.40 Klasse ListAssistantFrame 101
7.41 Interface ListAssistantListener 103
7.42 Klasse ListendarstellungFrame 103
7.43 Interface ListFrameListener 104
7.44 Klasse ListManager 104
7.45 Klasse LoginManager 106
7.46 Klasse RBManager 107
7.47 Klasse SaveFrame 109
7.48 Klasse SQLAssistant 109
7.49 Klasse SQLQuery 112
7.50 Klasse SuchErgebnis 113
7.51 Klasse SuchErgebnisCursor 116
7.52 Klasse TextColArea 116
7.53 Klasse TextColChoice 117
7.54 Klasse TextColField 119

7.55 Klasse TextColFieldwithList 119
7.56 Klasse TextColItem 120
7.57 Interface TextColListener 122

8 Besonderheiten in der Implementation von *Flit* 123
8.1 Klasse DokumentIndok 123
8.2 Klasse DokumentSchlagwoerter 124
8.3 Klasse ExprSchlagwortParser 126
8.4 Klasse Schlagwort 127
8.5 Klasse SchlagwortCanvas 129
8.6 Klasse SchlagwortFrame 130
8.7 Interface SchlagwortListener 132
8.8 Klasse Arbeitsablauf (Erweitert) 133
8.9 Klasse DokumentDatensatz (Erweitert) 133
8.10 Klasse SQLAssistant (Erweitert) 134
8.11 Klasse SuchErgebnis (Modifiziert) 135
8.12 ResourceBundle `Sprache.properties` (Erweitert) 135

9 Besonderheiten in der Implementation von *MetaMask* 137
9.1 eine andere Aufgabe 137
9.2 Klasse Arbeitsablauf 137
9.3 Klasse ChoiceData 138
9.4 Klasse ChoiceEditFrame 140
9.5 Interface ChoiceEditListener 140
9.6 Klasse DataManager 140
9.7 Klasse FieldEntry 147
9.8 Klasse RepositionFrame 149
9.9 Interface RepositionFrameListener 150
9.10 Klasse SQLAssistant 150
9.11 Klasse AblaufVerwalter (Erweitert) 151
9.12 Klasse FlitLoginFrame (Erweitert) 151

10 Besonderheiten in der Implementation von *Litera* **153**

10.1 die speziellen Anforderungen 153

10.2 Klasse BestellFrame 154

10.3 Interface BestellFrameListener 155

10.4 Klasse BestellListendarstFrame 156

10.5 Interface BestellListdarstListener 156

10.6 Klasse KontrollListManager 157

10.7 Klasse Warenkorb 157

10.8 Klasse Arbeitsablauf (Erweitert) 158

10.9 Klasse SQLAssistant (Erweitert) 159

10.10ResourceBundle `Sprache.properties` (Erweitert) 160

10.11Das erweiterte Rollensystem 160

Teil I

Benutzerhandbücher

Einleitung zum Programmpaket

Zu Beginn meiner Arbeit lag nur die von Simon Stelling programmierte Maske *Flit* für die Forschungsliteratur vor. Da diese sich als zuverlässig erwiesen hatte, bestand meine Aufgabe darin, mich in die Programmstruktur einzuarbeiten, die Maske auf die Literaturdatenbank zu portieren und schließlich eine Oberfläche zu programmieren, mit der *Flit*-ähnliche Masken für andere Datenbankrelationen erzeugt werden können. Dieser Maskengenerator liegt mit dem Programm *MetaMask* vor.

Die Masken werden jeweils für eine spezielle Datenbankrelation erzeugt und bieten von vornherein Grundfunktionalitäten für das Suchen und Arbeiten mit dieser Relation. Darüber hinaus können zusätzliche Funkitonalitäten in die erzeugten Klassen einprogrammiert werden.

Der vorliegende Text soll zunächst einen Überblick über die Benutzung der Programme *Flit* (die ursprüngliche Maske), *MetaMask* (den Maskengenerator) und *Litera* (eine Maske für die Literaturdatenbank, die mit *MetaMask* erzeugt wurde) geben. Da das Aussehen sämtlicher Benutzerschnittstellen dieses Programmpaketes sehr ähnlich ist, werden in Kapitel 1 zunächst die Elemente der Hauptfenster vorgestellt. In den Kapiteln 2 bis 4 werden dann die einzelnen Programme mit ihren Sonderfunktionalitäten vorgestellt.

Der zweite Teil befasst sich mit den Details der Realisation. In Kapitel 5 werden die Programm- und die Klassenstruktur erläutert, und in Kapitel 6 wird erklärt, wie eine Maske mit zusätzlichen Funktionen ausgestattet werden kann. Die Kapitel 8 bis 10 befassen sich mit den einzelnen Klassen der Programme *Flit*, *MetaMask* und *Litera*.

Kapitel 1

Die gemeinsamen Grundelemente der Benutzerschnittstellen

Die Hauptfenster der *Flit*-Maskenfamilie haben alle die gleichen Bereiche und die gleichen Grundelemente, aus denen sie aufgebaut sind. Dieselben Bestandteile sind auch im *MetaMask*-Hauptfenster zu finden, und obwohl sich *MetaMask* in mehreren Punkten von den erzeugten Masken unterscheidet, kann dieser Abschnitt auch als Einführung in die Benutzerschnittstelle für den Maskengenerator verstanden werden. Dennoch gelten die in diesem Kapitel erwähnten Details hauptsächlich für die erzeugten Masken; die Unterschiede zu *MetaMask* werden im Kapitel 3 genauer ausgeführt.

Die in diesem Kapitel enthaltenen Screenshots stellen die Fenster des Programms *Flit* vor.

1.1 Das Loginfenster

Wird eine Maske gestartet, so wird der Benutzer im Loginfenster (siehe Abbildung 1.1) zuerst um seinen Login in der Datenbank gebeten. Das Fenster hat eine Menüzeile, über die das Programm beendet, die Schriftgröße eingestellt oder ein Hilfetext angezeigt werden kann.

Gelingt der Login nicht, so bleibt das Loginfester sichtbar, und eventuelle Tippfehler können korrigiert werden. Ist der Login gelungen, so verschwindet das Loginfenster, und stattdessen erscheint das Maskenfenster.

Für Benutzer, die keinen eigenen Login haben, steht ein Gast-Login parat. Allerdings haben Gäste grundsätzlich kein Schreibrecht auf die Datenbankrelation (für die Verwaltung der Schreibrechte siehe Abschnitt 5.5).

Abbildung 1.1: Das Login-Fenster

1.2 Die Programmzustände

Die von *MetaMask* erzeugten Masken bieten von vornherein einige Funktionen. Dazu gehört das Suchen in der Datenbankrelation, das die Hauptanwendung darstellen dürfte. Der Zugriff auf die Datensätze der Relation ist nur über ein so erzeugtes Suchergebnis möglich, das allerdings noch erweitert und verfeinert werden kann.

Daneben bieten die Masken die Möglichkeit, neue Datensätze in die Relation einzufügen oder bereits vorhandene zu überarbeiten. Diese Funktionalität wird natürlich nur solchen Benutzern angeboten, die auch auf der Datenbankrelation Schreibrechte haben (siehe dazu auch Abschnitt 5.5).

Um diese Funktionen zu verwalten, haben die Masken drei verschiedene Programmzustände:

- **Kriterieneingabe** Wenn die Maske gestartet wird, befindet sie sich zunächst in diesem Zustand. Hier können Auswahlkriterien für die nachfolgende Suche eingegeben werden.

- **Anzeigen des Suchergebnisses** Wird eine Suche ausgeführt, so wechselt das Programm danach in diesen Zustand, in dem das Ergebnis angezeigt wird. Der erste Datensatz, der gefunden wurde, wird sofort im Hauptfenster dargestellt, während das Programm im Hintergrund weitere Datensätze aus der Relation ausliest. Es ist jedoch möglich, die angezeigten Datensätze sofort zu bearbeiten, auch wenn das Einlesen noch nicht abgeschlossen wurde.

- **Neueingabe** Dieser Zustand ist nur für diejenigen Benutzer zugänglich, die auf der Datenbankrelation Schreibrechte haben. Hier können neue Datensätze in die Relation eingefügt werden.

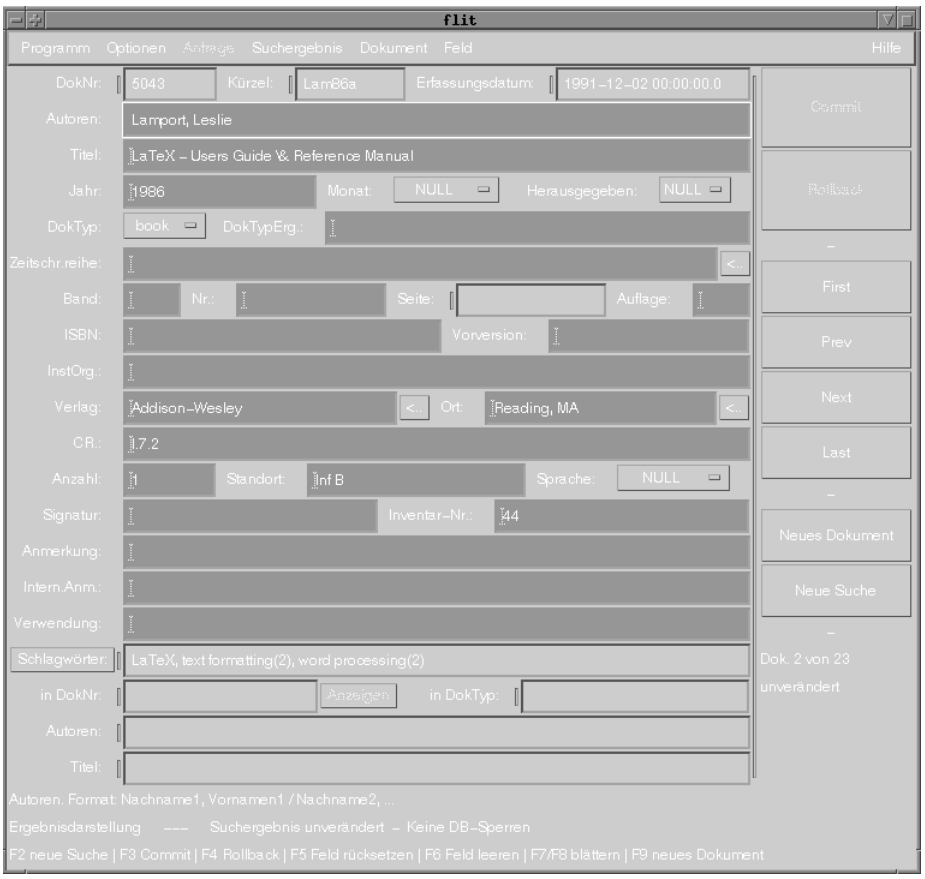

Abbildung 1.2: Das Hauptfenster

1.3 Das Maskenfenster

Das Hauptfenster der Masken (als Beispiel sei hier wieder *Flit* herangezogen, siehe Abbildung 1.2) besteht aus vier Bereichen:

- dem Hauptpanel, in dem der aktuelle Datensatz angezeigt wird,
- den Statuszeilen am unteren Fensterrand, in denen Hilfstexte und Programmmeldungen ausgegeben werden,
- der Knopfleiste am rechten Fensterrand, über die das Programm zu steuern ist und
- der Menüzeile, in der Programmeinstellungen vorgenommen werden können.

1.3.1 Das Hauptpanel

Das Hauptpanel ist der Bereich, in dem ein Datensatz angezeigt wird. Für jede Spalte der Datenbankrelation ist ein Feld vorhanden; außerdem können Knöpfe, die keine globale Funktion haben, im Hauptpanel (statt in der Knopfleiste) platziert werden.

Welche Funktionen oder Zusammenhänge diese Maskenelemente haben, kann selbstverständlich nicht für alle Masken einheitlich erklärt werden. Deshalb sollen hier lediglich die verschiedenen Typen angegeben werden, während die speziellen Funktionen bei den Programmen *Flit*, *MetaMask* und *Litera* in den Kapiteln 2 bis 4 aufgeführt sind.

- **Textfeld**
  Ein Textfeld bietet die Möglichkeit, beliebige Texte (oder Zahlen) einzugeben. Es ist weiß, wenn Eingaben angenommen werden; ansonsten ist es grau, um anzuzeigen, dass es deaktiviert ist.
- **Textfeld mit Liste**
  In einem Textfeld mit Liste ist es wie bei einem normalen Textfeld möglich, Daten einzugeben. Zusätzlich bietet der Knopf neben dem Textfeld ein Fenster, in dem aus den momentan in der Datenbank befindlichen Werte einer gewählt werden kann. Dies gibt dem Benutzer die einfache Möglichkeit, einen bereits vorhandenen Wert einzusetzen, aber auch neue Werte zu ergänzen.
- **Auswahlliste**
  Eine Auswahlliste bietet die Wahl zwischen mehreren festgelegten Werten. Auswahllisten werden an Stelle der Textfelder mit Listen benutzt, wenn es nicht sinnvoll ist, die Auswahl der Werte zu erweitern (z. B. bei Ja/Nein-Angaben).
- **Knopf**
  Knöpfe sind die einzigen Masken-Elemente, die sich nie auf eine Spalte der Datenbankrelation beziehen können. Stattdessen wird mit ihnen ein Kommando verknüpft, das an das Hauptprogramm weitergegeben wird.

Links neben jedem Maskenelement, das Werte anzeigt, ist ein kleiner Strich, der meistens nicht zu sehen ist. Dieser Strich zeigt (wenn er sichtbar ist) Statusinformationen für das zugehörige Feld an:

- er wird rot, um auf einen Fehler hinzuweisen. Genauere Informationen können den Statuszeilen entnommen werden.
- ist er schwarz, so wurde der Wert verändert
- sieht er aus, als ob er in die Maske eingestanzt wäre, so ist das Feld deaktiviert

1.3.2 Die Statuszeilen

Zu den Statuszeilen gehören sowohl die drei Zeilen am unteren Rand des Fensters als auch der kleinere Bereich unterhalb der Knopfleiste in der rechten unteren

Ecke des Fensters.

In den drei Zeilen am unteren Bildschirmrand werden Statusinformationen des Programms angezeigt, z. B. in welchem Zustand es sich gerade befindet. Ausserdem werden hier auch die Fehlermeldungen angezeigt und man kann hier ablesen, welche *Hotkeys* im Moment aktiviert sind; dies kann sich je nach Programmzustand ändern.

Im Bereich unter der Knopfleiste kann man sehen, ob der momentan angezeigte Datensatz modifiziert oder unverändert ist. Hier wird auch angezeigt, ob das Einlesen der Datensätze aus der Relation abgeschlossen ist oder nicht; für das Einlesen können zwei verschiedene Geschwindigkeiten ausgewählt werden – es ist sinnvoller, die Einlesegeschwindigkeit zu drosseln, wenn man in der Zwischenzeit bereits weiterarbeiten will.

1.3.3 Die Knopfleiste

Die Knopfleiste am rechten Fensterrand dient zur globalen Steuerung der Maske; hier kann von einem Zustand in den anderen gewechselt oder ein anderer Datensatz angezeigt werden. Die Knöpfe sind deaktiviert, wenn ihre Funktion im Moment nicht sinnvoll ist, wie z. B. das Ausführen der Suche, bevor Auswahlkriterien eingegeben wurden. Welche Knöpfe angezeigt werden, ändert sich mit den Programmzuständen; deshalb sollen diese Zustände hier einzeln aufgeführt werden:

im Zustand „Kriterieneingabe“

- **Ausführen** Dieser Knopf startet die Suche mit den momentan angezeigten Kriterien. Nach der Suche wird das Programm in den Zustand „Anzeigen des Suchergebnisses“ wechseln, falls die Suche mindestens einen Treffer ergeben hat.
- **Ergebnis erweitern** Auch mit diesem Knopf wird die Suche mit den angezeigten Kriterien gestartet, allerdings werden gefundene Datensätze zu dem eventuell schon vorhandenen Suchergebnis hinzugefügt.
- **Kriterien verfeinern** Hiermit wird eine Suche gestartet, die die angegebenen Kriterien nicht auf die Datenbank, sondern auf das vorher vorhandene Suchergebnis anwendet.
- **Felder leeren** Löscht sämtliche Einträge in den Textfeldern.
- **Letzte Kriterien** Holt die Kriterien zurück, die für die letzte Suche eingegeben wurden.
- **Neues Dokument** Wechselt in den Zustand „Neueingabe“.

im Zustand „Anzeigen des Suchergebnisses“

- **Änderungen übernehmen** Sind Änderungen am Suchergebnis vorgenommen worden, so können sie mit diesem Knopf in die Datenbank über-

tragen werden. Dies ist nur möglich, wenn der Benutzer Schreibrecht auf der Relation hat (siehe Abschnitt 5.5).

- **Änderungen rückgängig** Hiermit können eventuelle Änderungen am Suchergebnis wieder zurückgenommen werden. Die angezeigten Datensätze entsprechen dann wieder denen in der Datenbankrelation.
- **Erstes, Vorheriges, Nächstes, Letztes** Diese vier Knöpfe dienen dem Navigieren im Suchergebnis.
- **Neues Dokument** Wechselt in den Zustand „Neueingabe“.
- **Neue Suche** Wechselt in den Zustand „Kriterieneingabe“. Das Suchergebnis wird dadurch noch nicht gelöscht.

im Zustand „Neueingabe“

- **Änderungen übernehmen** Hiermit wird der neue Datensatz in die Relation eingefügt.
- **Änderungen rückgängig** Löscht den Inhalt aller Textfelder.
- **Vorherige Daten** Holt die Daten des letzten angezeigten Dokumentes in die Eingabemaske für das neue Dokument.
- **Neue Suche** Wechselt in den Zustand „Kriterieneingabe“.

1.3.4 Die Menüzeile

Die Menüzeile umfasst sieben Menüs, über die teilweise Parameter gesetzt werden und teilweise Operationen am aktuellen Datensatz durchgeführt werden können. Im einzelnen sind dies:

Menü Programm

- **Neues Fenster** Hiermit kann ein neues Maskenfenster geöffnet werden. Die Vorgänge in den unterschiedlichen Fenstern sind nicht miteinander verbunden.
- **Fenster schließen** Schließt das zugehörige Fenster. Wurden Änderungen vorgenommen, die noch nicht gesichert wurden, so wird der Benutzer vorher gefragt, ob diese in die Datenbank übertragen oder verworfen werden sollen.
- **Programm beenden** Beendet sämtliche offenen Maskenfenster dieses Programms. Wie beim Schließen eines Fensters wird vorher eine Abfrage auf Änderungen durchgeführt.

Menü Optionen

- **Expertenmodus** Wird der Expertenmodus eingeschaltet, so wird der Menüpunkt „SQL-Anfrage editierbar“ aktiviert. Darüber hinaus ist es möglich, einige Felder der Maske zusätzlich zu aktivieren. (Dies wurde z. B. bei *Flit* vorgenommen, siehe hierzu Kapitel 2).
- **SQL-Anfrage editierbar** Diese Option bewirkt, dass die Anfragen an die Datenbank vorher auf dem Bildschirm angezeigt werden und modifiziert werden können. Dies ermöglicht es, auch speziellere Anfragen zu stellen, die von der Maske nicht angeboten werden. Man muss allerdings darauf achten, dass die Anfrage gültig bleibt und das Rückgabeformat nicht geändert wird, da die Maske das Ergebnis sonst nicht erkennen kann; es wird keine weitere Konsistenzabfrage durchgeführt.

Menü Anfrage

- **Sortierung nach...** Es ist möglich, die Sortierung, in der das Suchergebnis von der Datenbank übergeben wird, zu bestimmen. Dafür muss aus der hier gegebenen Liste nur die gewünschte Kriterienabfolge ausgewählt werden. Da sich diese auf die Spalten der Relation beziehen, müssen sie allerdings bei neu erzeugten Masken separat eingegeben werden (siehe Kapitel 6).
- **Auto-Wildcards** Wird diese Option angeschaltet, so werden die in Textfeldern eingegebenen Werte am Anfang und Ende mit Wildcards versehen (außer bei einigen Textfeldern, bei denen dies explizit verboten ist). Benutzt wird hierfür das Zeichen ’%’.
- **Case-Sensitive** Diese Option bestimmt, ob in der Anfrage auf Groß- und Kleinschreibung geachtet werden soll.

Menü Suchergebnis

- **Im Voraus einlesen** Hier kann ausgewählt werden, ob bei einer Anfrage das Suchergebnis im Hintergrund eingelesen werden soll. Dies ist in zwei Geschwindigkeiten möglich – die langsamere empfiehlt sich, falls man die Datensätze schon bearbeiten will, bevor das Einlesen beendet ist.
- **Blättern mit Cursortasten** Wird diese Option angeschaltet, so kann man mit Hilfe der Cursortasten „hoch“ und „runter“ im Suchergebnis genau so navigieren wie mit den Knöpfen „Vorheriges“ und „Nächstes“. Ist sie ausgeschaltet, so springt man mit den Cursortasten über die verschiedenen aktivierten Textfelder des Hauptpanels.
- **Listenübersicht erzeugen** Diese Funktion ermöglicht es, das Suchergebnis in Listenform ausgeben zu lassen. Eine genauere Beschreibung findet sich im Abschnitt 1.4.

Menü Dokument

Die Funktionen dieses Menüs beziehen sich jeweils auf den gesamten momentan angezeigten Datensatz. Sie sind deaktiviert, wenn sie nicht anwendbar sind.

- **Löschen** Markiert den Datensatz als gelöscht. Er wird allerdings erst aus der Datenbank entfernt, wenn der Knopf „Änderungen übernehmen“ gedrückt wird.
- **Löschen rückgängig** Hiermit wird die Markierung, die durch den vorherigen Menüpunkt gesetzt wurde, wieder entfernt.
- **Änderungen rückgängig** Alle Änderungen, die an diesem Datensatz vorgenommen wurden, werden rückgängig gemacht. Andere Datensätze sind davon nicht betroffen.

Menü Feld

Die Funktionen diese Menüs beziehen sich lediglich auf ein einzelnes Textfeld, nämlich das, in dem der Fokus sich gerade befindet, bzw. das gerade angeklickt ist. Welches das ist, kann man den Statuszeilen entnehmen.

- **Vorheriger Inhalt** Stellt den Eintrag wieder her, der sich vorher in diesem Feld befunden hat.
- **Leeren** Löscht den Inhalt dieses Feldes.
- **Änderungen rückgängig** Nimmt alle Änderungen, die an diesem Feld vorgenommen wurden, wieder zurück. Andere Felder oder andere Datensätze sind davon nicht betroffen.

Menü Hilfe

- **Info** Zeigt einen Hilfetext an.
- **Version** Zeigt eine kurze Information über die Version der Maske an.

1.4 Der Listenassistent

Die erzeugten Masken bieten die Möglichkeit, Suchergebnisse in Listenform ausgeben zu lassen. Diese Funktion wird über das Menü Suchergebnis, Menüpunkt „Listenübersicht erzeugen“ angesprochen.

Wird sie aktiviert, so wird zunächst ein separates Fenster geöffnet, der Listenassistent (Abbildung 1.3). Hier kann eingeben werden, welche Spalten der Relation in welcher Reihenfolge ausgegeben werden sollen, und durch welche Zeichen (oder Texte) die Ausgabe ergänzt wird. Diese Angabe kann auch aus einer Liste ausgewählt werden. Da diese Listen allerdings von der Relation abhängen, müssen sie bei einer neu erzeugten Maske separat eingegeben werden (siehe dazu

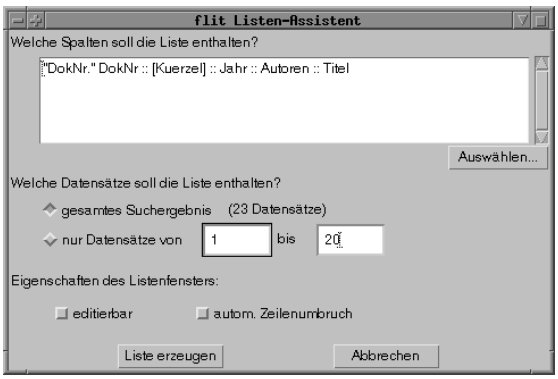

Abbildung 1.3: Der Listenassistent

Kapitel 6). Die Angabe der Spalten im Listenassistenten erfolgt duch Eintragen des Namens oder zugeordneten Kürzels. Die erzeugte Liste wird ein einem Listenfenster angezeigt (siehe Abbildung 1.4).

flit Listenübersicht
flit Listenübersicht
Format: "DokNr." DokNr :: [Kuerzel] :: Jahr :: Autoren :: Titel
DokNr. 5037 :: [Kop90a] :: 1990 :: Kopka, Helmut :: LaTeX – Eine Einf"uhrung (2. Auflage)
DokNr. 5043 :: [Lam86a] :: 1986 :: Lamport, Leslie :: LaTeX – Users Guide \& Reference Manual
DokNr. 5230 :: [Kop91a] :: 1991 :: Kopka, Helmut :: LaTeX – Erweiterungsm"oglichkeiten
DokNr. 7253 :: [Kop91b] :: 1991 :: Kopka, Helmut :: LaTeX – Eine Einf"uhrung (4. Aufl.)
DokNr. 7986 :: [PSH90a] :: 1990 :: Partl, Hubert / Schlegl, Elisabeth / Hyna, Irene :: LaTeX–Kurzbeschreib
DokNr. 7987 :: [BHR89a] :: 1989 :: Braune, Klaus / Heinrich, Dietmar / Rupprecht, Rainer :: Ein LaTeX–K
DokNr. 9382 :: [DL94a] :: 1994 :: Dietsche, L. / Lammarsch, J. :: LaTeX zum Loslegen – Ein Soforthelfer f
DokNr. 10084 :: [Lam94a] :: 1994 :: Lamport, Leslie :: A Document Preparation System LaTeX -- User's
DokNr. 10085 :: [Kop94a] :: 1994 :: Kopka, Helmut :: LaTeX -- Einf"uhrung, Band 1
DokNr. 10086 :: [Kop94b] :: 1994 :: Kopka, Helmut :: LaTeX -- Erg"anzungen, Band 2
DokNr. 10087 :: [Kop97a] :: 1997 :: Kopka, Helmut :: LaTeX -- Erweiterungen, Band 3
DokNr. 10277 :: [GMS94a] :: 1994 :: Goossens, Michel / Mittelbach, Frank / Samarin, Alexander :: The La
DokNr. 10338 :: [Sch94e] :: 1994 :: Schwarzkopf, Otfried :: The Hyperlatex Markup Language
DokNr. 10855 :: [EH95a] :: 1995 :: Esser, Thomas / Hillbrecht, Dirk :: Integration von \LaTeX\ in eine Umg
DokNr. 11033 :: [SP97a] :: 1997 :: Schwarz, Stefan / Potucek, Rudolf :: Das {\TeXikon} -- Referenzhandb
DokNr. 11042 :: [GMS95a] :: 1995 :: Goossens, Michel / Mittelbach, Frank / Samarin, Alexander :: Der La
DokNr. 11553 :: [Lam94c] :: 1994 :: Lamport, Leslie :: A Document Preparation System LaTeX -- User's
Schließen

Abbildung 1.4: Das Listenfenster

Man hat die Möglichkeit, entweder das gesamte Suchergebnis als Liste ausgeben zu lassen, oder nur einen bestimmten Bereich, der mittels des ersten und letzten gewünschten Datensatzes angegeben werden kann.

Es können auch die Eigenschaften „editierbar“ und „automatischer Zeilenumbruch“ für das Listenfenster ausgewählt werden. Ist das Fenster editierbar, so kann der Benutzer weitere Änderungen an der Listenausgabe vornehmen, falls er dies als notwendig erachtet. Soll kein automatischer Zeilenumbruch vorgenommen werden, so werden die Daten eines Dokumentes in einer einzigen Zeile ausgegeben, und erst für das nächste Dokument wird eine neue Zeile begonnen.

Kapitel 2

Flit Benutzerhandbuch

2.1 Übereinstimmung mit den Grundfunktionen

Flit ist die ursprüngliche, von Simon Stelling programmierte Maske für die Datenbankrelation der Forschungsliteratur. Der Großteil der Funktionalität ist identisch mit den Grundelementen, die schon im Kapitel 1 vorgestellt wurden.

In der Forschungsliteraturdatenbank werden allerdings auch Schlagwörter zu den einzelnen Dokumenten gespeichert, und *Flit* besitzt daher zusätzliche Funktionen und Textfelder, um dem gerecht zu werden. Außerdem ist die Möglichkeit gegeben, dass sich einzelne Dokumente in anderen Dokumenten befinden, wie z. B. Artikel, die in Büchern enthalten sind. Auch hierfür waren zusätzliche Textfelder notwendig.

Die einzelnen Maskenelemente werden im folgenden Abschnitt beschrieben, während auf die Behandlung der Schlagwörter in Abschnitt 2.3 eingegangen wird.

2.2 Beschreibung der Maskenelemente

Das Hauptfenster der Maske *Flit* wurde schon in Abbildung 1.2 dargestellt; für besseres Verständnis der folgenden Beschreibung sei nochmals auf diese verwiesen.

Im folgen Abschnitt sollen der Vollständigkeit halber alle Maskenelemente kurz erwähnt werden.

- **DokNr** die laufende Nummer des angezeigten Dokumentes
- **Kürzel** das Kürzel, unter dem das Dokument eingeordnet ist; es enthält üblicherweise eine Abkürzung des Autoren und das Erscheinungsjahr

- **Erfassungsdatum** wird automatisch von der Datenbank eingesetzt und enthält die Angabe, wann dieses Dokument zum ersten Mal eingefügt wurde
- **Autoren** Angabe des oder der Autoren
- **Titel** Angabe des Dokumententitels; dieses Feld darf nicht leer gelassen werden
- **Jahr** vierstellige Angabe des Erscheinungsjahres
- **Monat** Angabe des Erscheinungsmonats
- **Heraugegeben** gibt an, ob die Autoren nur Herausgeber sind
- **DokTyp** diese Auswahlliste gibt an, welchen Typ das angegebene Dokument hat; von der Angabe in diesem Feld ist abhängig, ob einige Felder deaktiviert sind, oder ob ein enthaltendes Dokument angegeben werden kann
- **DokTypErg** Ergänzungen zum Dokumententyp
- **Zeitschriftenreihe** falls es sich bei dem Dokument um eine Zeitschrift handelt, kann hier die entsprechende Reihe angegeben werden
- **Band** Band der Zeitschrifenreihe
- **Nr** Nummer der Zeitschrift
- **Seite** Seitenzahlen bei Teildokumenten in der Form „erste Seite - letzte Seite“
- **Auflage** Auflage des Dokuments
- **ISBN** ISBN des Dokuments
- **Vorversion** Dokumentnummer der Vorversion
- **InstOrg** Name der Organisation, der die Autoren angehören
- **Verlag** der Verlag, in dem das Dokument veröffentlicht wurde
- **Ort** Standort des Verlages
- **CR** CR-Klassifikation
- **Anzahl** in welcher Anzahl das Dokument vorhanden ist
- **Standort** wo das Dokument steht
- **Sprache** in welcher Sprache das Dokument verfasst ist
- **Signatur** Signatur, unter der das Dokument zu finden ist
- **Inventar-Nr** die Inventarnummer des Dokuments
- **Anmerkung** in diesem Feld können beliebige, öffentlich zugängliche Angaben hinzugefügt werden
- **Intern.Anm** Im Gegensatz zu dem Text, der im Feld „Anmerkung“ eingegeben ist, werden die internen Anmerkungen nicht an Benutzer weitergegeben, die sich mittels des Gast-Logins angemeldet haben.

- **Verwendung** Angaben über die Verwendung des Dokuments
- **Schlagwörter** In diesem Textfeld werden die Schlagwörter des Dokumentes angezeigt, falls solche vorhanden sind. Gleichzeitig ist dieser Bezeichner ein Knopf, der ein separates Fenster öffnet, in dem die Schlagwörter modifiziert oder neue hinzugefügt werden können; für eine genauere Beschreibung siehe Abschnitt 2.3.
- **in DokNr** Ist das angezeigte Dokument in einem anderen enthalten, so wird hier die Dokumentennummer des anderen Dokumentes angezeigt. Die Informationen, um welchen Dokumententyp es sich bei dem umfassenden Dokument handelt, sowie dessen Autoren und Titel werden in den nachfolgenden Textfeldern angezeigt. Diese Felder sind immer deaktiviert, da Veränderungen an diesen Werten in dem umfassenden DokumentDatensatz vorgenommen werden sollten.
- **Anzeigen** Dieser Knopf bezieht sich ebenfalls auf das Dokument, in dem das aktuelle enthalten ist. Durch Drücken kann man ein neues Maskenfenster öffnen, das dieses andere Dokument anzeigt.

2.3 Schlagwörter

Im Unterschied zu der Grundfunktionalität der erzeugten Masken umfasst die Forschungsliteraturdatenbank nicht nur eine einzige Relation: in einer weiteren Relation sind Schlagwörter zu den Dokumenten abgelegt, die mit unterschiedlichen Gewichten versehen werden können. *Flit* besitzt daher Erweiterungen, um auf diese zuzugreifen.

Wird der Knopf „Schlagwörter“ im *Flit*-Hauptfenster gedrückt, so wird ein neues Fenster geöffnet, das die Schlagwörter des aktuellen Dokumentes anzeigt (Abbildung 2.1). Hier ist auch angegeben, welches Gewicht das Schlagwort besitzt, und von wem es eingetragen wurde.

Mittels der Felder im unteren Fensterbereich können Schlagwörter modifiziert und zu den vorhandenen hinzugefügt werden. Die Menüleiste bietet drei Einträge mit den folgenden Funktionen:

- **Fenster**
  - **Fenster schließen** beendet das Schlagwortfenster und gibt die Programmkontrolle an das Hauptfenster zurück; dies ist auch mit dem *Hotkey* „CTRL-W“ möglich.
- **Schlagwort** Die Menüeinträge dieses Menüs beziehen sich jeweils auf ein einziges Schlagwort. Im einzelnen sind dies:
  - **Neues Schlagwort** fügt ein neues Schlagwort hinzu – diese Funktion entspricht dem Knopf im unteren Teil des Fensters und kann auch durch „CTRL-N“ erreicht werden.
  - **Löschen** löscht das Schlagwort aus der Liste („CTRL-D“).

Abbildung 2.1: Das Schlagwortfenster

  - **Löschen rückgängig** fügt ein gelöschtes Schlagwort wieder zu der Liste hinzu. Dies ist nur solange möglich, bis im Hauptfenster „Commit“ gedrückt wird.
  - **Änderungen rückgängig** setzt dieses Schlagwort wieder auf seinen ursprünglichen Zustand zurück, lässt aber die anderen eventuell modifiziert.

- **Schlagwörter**
  - **Alles rückgängig für diese DokNr** nimmt sämtliche Änderungen und Löschungen in der Schlagwortliste des aktuellen Dokuments zurück.

2.4 umfassende Dokumente

In der Forschungsliteraturdatenbank ist vorgesehen, dass ein Dokument in einem anderen enthalten sein kann. Dies wird im Hauptfenster in den unteren drei Zeilen angegeben. Die Felder in diesen Zeilen (namentlich „in DokNr“, „in DokTyp“, „Autoren“ und „Titel“) geben die entsprechenden Informationen über das umfassende Dokument an. Es ist zu bemerken, dass sie immer deaktiviert sind, weil Änderungen an diesen Daten in dem umfassenden Dokument selbst vorgenommen werden sollen.

Die vollständigen Informationen zum umfassenden Dokument können mittels des Knopfes „Anzeigen“ aufgerufen werden. *Flit* öffnet dabei ein neues Hauptfenster, das dieses Dokument enthält. Das neue Fenster ist allerdings im Programmablauf unabhängig von dem ersten und diesem nicht etwa untergeordnet.

Kapitel 3

MetaMask Benutzerhandbuch

Das Programm *MetaMask* ist keine Datenbankmaske im eigentlichen Sinne der *Flit*-Familie, sondern der Generator für solche Masken. Obwohl das Hauptfenster dem des Programmes *Flit* sehr ähnlich sieht, gibt es eine beachtliche Zahl von Unterschieden.

Von vornherein sollte klar sein, dass die „Datensätze“, die von *MetaMask* angezeigt und bearbeitet werden, die Maskenelemente (also Textfelder, Knöpfe etc.) der erzeugten Maske sind. Dementsprechend werden wir in diesem Kapitel nicht von Datensätzen, sondern stattdessen von Maskenelementen sprechen. Verwechslungen mit den Maskenelementen des *MetaMask*-Hauptpanels sind nicht zu befürchten – da die Typen die selben sind, die in Kapitel 1 beschrieben wurden, bedürfen sie keiner weiteren Erwähnung.

3.1 Das Loginfenster

MetaMask braucht nur ein einziges Mal auf die Datenbank zuzugreifen: wenn nämlich die Arbeit an einer neuen Relation begonnen wird. Ab dann werden sämtliche Informationen über die Maskenelemente der zu erzeugenden Maske im Speicher behalten, und keine weiteren Daten sind notwendig. Soll ein bereits existierendes Projekt weiter bearbeitet werden, so ist daher kein Zugriff auf die Datenbank mehr nötig.

Das Startfenster (siehe Abbildung 3.1) bietet zwei Möglichkeiten, die Metadaten über die Maskenelemente einzulesen: einerseits kann eine Datenbankverbindung aufgebaut werden, aus der die Beschreibung einer Relation geholt wird, andererseits kann der Pfad einer bereits existierenden Maske der *Flit*-Familie angegeben werden, aus deren Files die Informationen extrahiert werden können.

In beiden Fällen muss ein Name für die Maske und der Wurzelpfad (siehe dazu Abschnitt 5.1) angegeben werden.

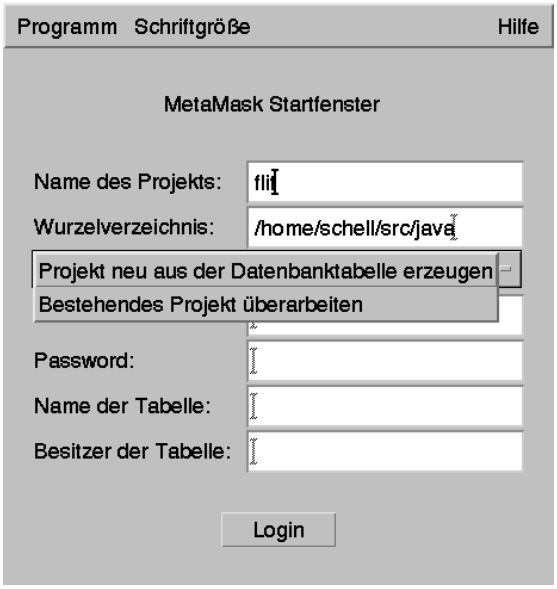

Abbildung 3.1: Das *MetaMask*-Startfenster

3.1.1 Laden aus der Datenbank

Dieses Verfahren ist nur notwendig, wenn für die in Frage stehende Tabelle noch keine Maske erzeugt wurde. Wird diese Option ausgewählt, falls schon eine Maske des selben Namens existiert, so werden beim Speichern die Einstellungen und gegebenenfalls Erweiterungen der alten Maske überschrieben.

Soll die Tabellenbeschreibung aus der Datenbank ausgelesen werden, so muss selbstverständlich eine Verbindung zu dieser aufgebaut werden; dafür werden ein Loginname und das dazugehörige Passwort für die Datenbank benötigt. Außerdem muss der Name der zu betrachtenden Tabelle angegeben werden. Es ist darüber hinaus möglich, einen Namen für den Besitzer der Tabelle anzugeben, falls dieser nicht mit dem Loginnamen übereinstimmen sollte; wird dieses Feld leer gelassen, so wird automatisch der angegebene Loginname als Tabellenbesitzer übernommen.

Treten beim Einloggen in die Datenbank Fehler auf (etwa weil das Passwort nicht akzeptiert wird), so können Korrekturen an den Eingaben vorgenommen und der Login-Vorgang wiederholt werden.

Da beim Laden aus der Datenbank noch kein Layout definiert wurde, generiert *MetaMask* ein neues Layout derart, dass immer drei Maskenelemente in einer Zeile angeordnet sind. Dass dabei eventuell Maskenelemente ausserhalb des Fensters platziert werden können stellt kein Problem dar, da dieses *Default*-Layout ohnehin noch mit *MetaMask* überarbeitet werden muss.

3.1.2 Lesen von vorhandenen Masken

Findet *MetaMask* bereits eine Maske mit dem angegebenen Namen im Pfad vor, so werden die vier Eingabefelder, die für den Login zur Datenbank gebraucht werden, deaktiviert, und *MetaMask* geht davon aus, dass die bestehende Maske überarbeitet werden soll. In diesem Fall brauchen keine weiteren Angaben gemacht zu werden.

Soll eine bestehende Maske überschrieben werden, so kann dies geschehen, indem man das Auswahlfeld auf „aus der Datenbank erzeugen“ umstellt. Hierbei werden allerdings sämtliche Veränderungen oder Erweiterungen der alten Maske verworfen!

3.2 Die Programmzustände

Da *MetaMask* nicht mehr auf die Datenbank zuzugreifen braucht, während eine Maske bearbeitet wird, können sämtliche benötigten Daten im Speicher gehalten werden. Der Programmzustand der Kriterieneingabe ist damit nicht mehr notwendig und entfällt. *MetaMask* kommt also mit lediglich zwei Programmzuständen aus, „Anzeigen aller Maskenelemente“ (enspricht „Anzeigen des Suchergebnisses“) und „Neueingabe“. Die Funktionen dieser Zustände können direkt aus Kapitel 1 übertragen werden.

3.3 Das *MetaMask*-Hauptfenster

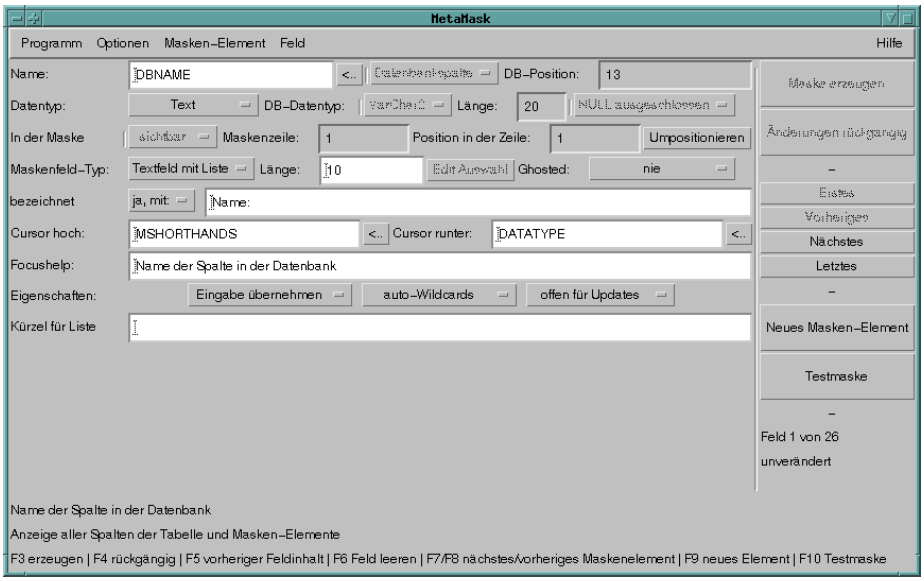

Abbildung 3.2: Das *MetaMask*-Hauptfenster

Das Hauptpanel (siehe Abbildung 3.2) besteht aus den selben Maskenelementen wie das der erzeugten Masken – für deren Beschreibung kann daher auf Abschnitt 1.3.1 verwiesen werden.

Die Statuszeilen entsprechen ebenfalls denen der erzeugten Masken, bis auf den Unterschied, dass die Informationen über das Einlesen der Daten entfallen, denn diese werden komplett im Speicher gehalten und brauchen nicht aus der Datenbank nachgeladen zu werden.

Die Menüzeile ist eine verkleinerte Fassung der Menüzeile wie sie in Abschnitt 1.3.4 beschrieben wurde. Die Funktionen, die entfielen, sind hauptsächlich diejenigen, die sich auf das Suchen in der Datenbank beziehen. Darüberhinaus ist die Funktionalität des Menüs „Dokument“ auf das Menü „Maskenelement“ zu übertragen. Allerdings ist es nicht möglich, Elemente zu löschen, die zu Spalten in der Datenbankrelation gehören; diese können aber als „in der Maske unsichtbar“ deklariert werden, sollte dies gewünscht sein.

Besondere Erwähnung finden sollen hier die drei Möglichkeiten, auf ein anderes Maskenelement zu wechseln. Neben den Navigationsknöpfen in der Knopfleiste, die denen der *Flit*-ähnlichen Masken entsprechen, kann man auch im Feld „Name“ des Hauptpanels die Bezeichnung eines Maskenelements eingeben, zu dem dadurch gesprungen wird (siehe auch Abschnitt 3.5, in dem das *MetaMask*-Hauptpanel detailliert beschrieben wird), und wenn eine Testmaske offen ist, so kann ein Maskenelement auch ausgewählt werden, indem man es in der dieser anklickt (siehe Abschnitt 3.4 über die Testmaske).

3.3.1 Die Knopfleiste

Die Knopfleisten für die zwei Programmzustände sehen denen der erzeugten Masken ebenfalls sehr ähnlich – doch gibt es ein paar feine Unterschiede, weshalb deren Funktionen hier dennoch einzeln aufgeführt werden sollen.

im Zustand „Anzeigen aller Maskenelemente“

- **Maske erzeugen** Mit diesem Knopf werden die Klassen für die Maske geschrieben. Sie stehen danach für Überarbeitungen und Erweiterungen von Hand zur Verfügung – oder auch direkt zum Compilieren.
- **Änderungen rückgängig** Hiermit können sämtliche Veränderungen an allen Maskenelementen zurückgenommen werden. Existierte bereits eine Maske, so entsprechen die angezeigten Daten wieder dieser Maske, ansonsten werden alle Felder auf die Defaultwerte zurückgesetzt.
- **Erstes, Vorheriges, Nächstes, Letztes** Von der Funktion entsprechen diese Knöpfe genau denen in den erzeugten Masken. Die Reihenfolge, auf die sie sich beziehen, ist die durch die zeilenweise Anordnung in der erzeugten Maske gegebene.
- **Neues Maskenelement** Wechselt in den Zustand „Neueingabe“.

- **Testmaske** Mit diesem Knopf wird eine neue Testmaske geöffnet, bzw. eine bereits offene auf den aktuellen Stand gebracht. Für eine genauere Beschreibung siehe Abschnitt 3.4.

im Zustand „Neueingabe“

- **Element einfügen** Fügt das neue Maskenelement in die Maske ein und wechselt in den Zustand „Anzeigen aller Maskenelemente“.
- **nicht einfügen** Dieser Knopf wechselt in den Zustand „Anzeigen aller Maskenelemente“ zurück, ohne das Element einzufügen. Sämtliche eingegebene Informationen (auch die für eventuelle Auswahllisten) werden verworfen.
- **Leeren** Löscht alle Felder. Auch die eventuell eingegebenen Daten einer Auswahlliste werden gelöscht.

3.4 Die Testmaske

Die Testmaske ist hauptsächlich dafür gedacht, das Layout der erzeugten Maske zu überprüfen. Eine Testmaske, die geöffnet oder aktualisiert wird, orientiert sich immer an den momentanen Werten der Maskenelemente; es ist also nicht notwendig, vorher auf „Maske erzeugen“ im Hauptfenster zu klicken – vielmehr kann dies dazu führen, dass die bisherige Fassung überschrieben wird und nicht wieder hergestellt werden kann, selbst wenn die neue sich als weniger gut herausstellt.

Da das erzeugen des Hauptpanels in der Testmaske eine relativ aufwendige Operation ist, wird die Testmaske nicht bei jeder Änderung aktualisiert. Sollen die vorgenommenen Änderungen übernommen werden, so muss der Knopf „Testmaske aktualisieren“ im der Knopfleiste des Hauptfensters angeklickt werden.

Obwohl die Testmaske der erzeugten Maske genau entspricht, hat sie keinerlei Funktionalität. Das Anklicken von Feldern, Knöpfen oder Menüeinträgen hat lediglich zur Folge, dass das entsprechende Kommando, das an das Hauptprogramm weitergegeben werden würde, in den Statuszeilen des Hauptfensters ausgegeben wird. Eine Ausnahme davon bilden die Menüeinträge „Fenster schliessen“ und „Programm beenden“, die beide dazu führen, dass die Testmaske wieder geschlossen wird.

Darüber hinaus kann die Testmaske genutzt werden, um zwischen den Maskenelementen zu wechseln. Wird nämlich ein Feld der Testmaske angeklickt, so zeigt das *MetaMask*-Hauptfenster den dazugehörigen Eintrag an. In der Testmaske kann auch festgestellt werden, ob die Auswahllisten richtig eingegeben wurden.

3.5 Bedeutungen der einzelnen Felder des Hauptpanels

Wie bereits erwähnt, besteht ein angezeigter Datensatz aus denjenigen Daten, die ein Maskenelement der erzeugten Maske beschreiben. Es gibt zwei verschiedene Arten von solchen Maskenelementen: einerseits kann eine direkte Entsprechung in der Datenbankrelation bestehen, andererseits können die Elemente nachträglich hinzugefügt worden sein und daher keine solche Entsprechung besitzen, wie das z. B. bei Knöpfen notwendigerweise der Fall ist.

Die Elemente des *MetaMask*-Hauptpanels sind in drei Gruppen sortiert:

- die Informationen über die Spalte in der Relation: Datentyp, Länge etc.
- das Layout in der erzeugten Maske und
- Texte für den Benutzer der erzeugten Maske.

Die Funktionen der einzelnen Elemente werden in den folgenden Abschnitten beschrieben.

datenbezogene Informationen

Bis auf die ersten beiden haben die Felder dieses Abschnittes bei Maskenelementen, die sich nicht auf Spalten der Relation beziehen, keine Bedeutung.

- **Name** Der Name eines Maskenelementes ist vergleichbar mit einem Primärschlüssel: hierüber erfolgt der eindeutige Zugriff auf dieses Element. Diese Feld enthält für Spalten in der Relation deren Spaltennamen, bei anderen Maskenelementen (z. B. Knöpfen) den Befehl, der bei Betätigen an das Hauptprogramm weitergegeben wird. Durch Eintragen eines Namens in dieses Feld können andere Maskenelemente angezeigt werden; es ist jedoch nicht möglich, dadurch neue Felder hinzuzufügen, dazu dient der Knopf „Neues Maskenelement“.
- **Datenbankspalte / Layoutelement** Dieses Feld, das nie editierbar ist, zeigt an, ob es sich bei dem aktuellen Maskenelement um eines handelt, das eine Entsprechung in der Datenbankrelation besitzt, oder ob es ein vom Benutzer hinzugefügtes Element ist.
- **DB-Position** Hier ist angegeben, welche Nummer die gerade angezeigte Spalte besitzt. Bei neuen Maskenelementen ist dieses Feld leer.
- **Datentyp** Diese Auswahlliste setzt fest, auf welchen Datentyp eine Eingabe getestet werden soll.
- **DB-Datentyp** Der Datentyp, den die Spalte beinhaltet, ist hier angegeben.
- **Länge** Diese Länge ist diejenige, die in der Datenbank für diese Spalte angegeben ist.
- **NULL ausgeschlossen / darf NULL sein** Je nach dem, ob die Datenbank den Wert NULL in dieser Spalte zulässt, wird diese Anzeige gesetzt.

Beschreibung des Layouts

- **sichtbar / verborgen** Für Datenbankspalten ist hier die Möglichkeit gegeben, sie aus der Maske auszublenden. Wird dieser Wert auf „verborgen“ gesetzt, so werden sämtliche Layout-Informationen ignoriert. Es ist nur möglich, solche Maskenelemente auszublenden, bei denen der Wert nicht durch die Auswahl „Eingabe übernehmen“ erzeugt wird (vergleiche den Abschnitt über dieses Feld weiter unten). Außerdem können vom Benutzer ergänzte Maskenelemente nicht ausgeblendet werden – für diese ist lediglich das Entfernen durch Löschen möglich.
- **Maskenzeile, Position in der Zeile** Diese beiden Felder geben die Position des Maskenelements in der erzeugten Maske an. Es ist nicht möglich, in diese Felder Werte einzutragen; dafür steht der folgende Knopf zur Verfügung:
- **Umpositionieren** Dieser Knopf öffnet ein Fenster, in dem die neue Position in der Maske angegeben werden kann; für eine genauere Beschreibung sei auf Abschnitt 3.6 verwiesen.
- **Element-Typ** Hier kann angegeben werden, ob das Maskenelement ein Textfeld, ein Textfeld mit Liste, eine Auswahlliste oder ein Knopf sein soll. Für die Beschreibung dieser Typen siehe Abschnitt 1.3.1.
  Ein Textfeld mit Liste ist einem solchen ohne Liste vorzuziehen, wenn Werte, die in der Relation auftreten, häufig wieder verwendet werden sollen. Sind sämtliche zulässigen Werte von vornherein bekannt, so sollte eine Auswahlliste benutzt werden.
  Je nach Auswahl in diesem Feld können die beiden folgenden Felder deaktiviert sein.
- **Länge** Hier ist die Länge des Feldes in der erzeugten Maske angegeben. Wird ein neues Projekt begonnen, so wird dieser Wert auf die Länge in der Datenbank gesetzt; dies ist jedoch nicht immer sinnvoll (z. B. bei Spalten, die lange Texteingaben akzeptieren), deshalb kann dieser Eintrag verändert werden.
- **Edit Auswahl** Mit diesem Knopf kann die Auswahlliste editiert werden. Eine detailliertere Beschreibung findet sich in Abschnitt 3.7.
- **deaktiviert** Sollen einige Maskenelemente je nach Programmzustand deaktiviert sein, so kann dies hier eingestellt werden. Angeboten werden alle möglichen Kombinationen der drei Programmzustände „Neueingabe“, „Kriterieneingabe“ und „Ergebnisdarstellung“.
- **Eingabe übernehmen / vom Programm erzeugt / als NULL einfügen** Mit dieser Auswahl kann entschieden werden, ob beim Erzeugen eines neuen Datensatzes ein Wert aus der Maske ausgelesen werden soll, ob das Programm eine Routine besitzt, die einen neuen Wert erzeugt, oder ob einfach der Wert NULL eingesetzt werden soll. Letzteres ist insbesondere dann vorteilhaft, wenn die Datenbank (mittels eines Datenbank-Triggers) bei der Eingabe NULL einen neuen Wert selbst erzeugt. Wie und wo die erzeugte Maske modifiziert werden muss, damit es selbst Werte erzeugen kann, wird in Kapitel 6 beschrieben.

- **auto-Wildcards / keine auto-Wildcards** Bei einigen Textfeldern (z. B. Buchtiteln) ist es sinnvoll, die Eingabe in Wildcards einzufassen. Dieses Feld verhindert dies generell; doch auch wenn hier „auto-Wildcards“ ausgewählt wird, können in der erzeugten Maske Anfragen ohne Wildcards gestellt werden (siehe auch Abschnitt 1.3.4).
- **offen für Updates / kein Update möglich** Soll ein einmal eingetragener Wert unveränderlich bleiben (etwa bei einer Inventarnummer), so kann das in diesem Feld entsprechend eingestellt werden.
- **bezeichnet** Dies ist eine Kombination aus zwei Feldern: einer Auswahlliste mit den Werten „ja, mit:“ und „nein.“, sowie einem dazugehörigen Textfeld. Bei Textfeldern und Auswahllisten bezieht sich diese Angabe auf einen Textbezeichner, der vor das Maskenelement geschrieben wird (wie bei den meisten Feldern im *MetaMask*-Hauptpanel – als Beispiel für eine Auswahlliste ohne separaten Bezeichner sei das vorangegangene Feld (Updates) genannt). Bei Knöpfen wird hier die Beschriftung angegeben.
- **Cursor hoch, Cursor runter** Wird im Menü „Optionen“ die Einstellung „Blättern mit Cursortasten“ ausgeschaltet, so kann hier angegeben werden, welches Feld bei einem entsprechenden Tastendruck den Fokus erhält.

Texte

- **Focushelp** Der Text in diesem Feld erscheint in des Statuszeilen der erzeugten Maske, wenn das Maskenelement den Fokus erhält. Hier kann eine kurze Beschreibung der Funktion des entsprechenden Elements angegeben werden, die dem Benutzer helfen soll.
- **Kürzel für Liste** Hier können Kürzel eingetragen werden, die im Listenassistenten diese Spalte bezeichnen. Der Listenassistent wird in Abschnitt 1.4 separat beschrieben.

3.6 Das Umpositionieren von Maskenelementen

Maskenelemente können nicht einfach im Hauptpanel der erzeugten Maske verschoben werden, indem neue Werte für „Maskenzeile“ oder „Position in der Zeile“ eingegeben werden; durch solche Eintragungen könnte es vorkommen, dass Positionen doppelt oder gar nicht besetzt wären. Stattdessen befindet sich im *MetaMask*-Hauptpanel der Knopf „Umpositionieren“, der ein separates Fenster öffnet.

In diesem Fenster kann nun die gewünschte neue Position eingetragen werden. Es gibt drei Möglichkeiten, dieses zusätliche Fenster wieder zu schließen:

- **alte Position behalten** In diesem Fall bleibt die bisherige Anordnung unverändert.

- **neue Position annehmen** Dies führt dazu, dass das aktuelle Maskenelement an die neue Position eingefügt wird, und zwar vor einem Maskenelement, das sich eventuell schon an dieser Position befindet.
- **als neue Zeile einfügen** Wird dieser Knopf gedrückt, so wird das aktuelle Maskenelement als erstes Element einer neuen Zeile eingefügt. Alle dahinter befindlichen Zeilen werden um eine Zeile nach unten verschoben.

Eine eventuell geöffnete Testmaske wird die neue Position allerdings nicht automatisch annehmen; dafür muss der Knopf „Testmaske aktualisieren“ im Hauptfenster angeklickt werden.

3.7 Das Editieren von Auswahllisten

Ist das aktuelle Maskenelement eine Auswahlliste, so ist im Hauptpanel der Knopf „Edit Auswahlliste“ aktiviert. Wird dieser angeklickt, so wird ein separates Fenster geöffnet, das den momentanen Inhalt der Auswahlliste enthält. Dieser kann dann beliebig erweitert und verändert werden.

In diesem Fenster werden abwechselnd eine Zeile erwartet, die den Wert oder Text beinhaltet, der in der erzeugten Maske angezeigt werden soll, und eine Zeile mit dem Wert, der in der Datenbankrelation benutzt wird. Führende und angehängte Leerzeichen werden dabei beachtet.

Es wird empfohlen, nach einem solchen Paar Zeilen eine Zeile Abstand zu halten; Leerzeilen in der Eingabe werden ohnehin komplett ignoriert.

Das Fester kann mit den beiden Knöpfen „Änderungen übernehmen“ und „Änderungen verwerfen“ geschlossen werden, mit denen entweder die neue Eingabe gespeichert bzw. der ursrpüngliche Zustand wiederhergestellt wird.

Kapitel 4

Litera Benutzerhandbuch

Litera ist eine Instantiierung der Template-Klassen für die Literaturdatenbank. Der Großteil der Funktionalität entspricht den Grundfunktionen, die schon im Kapitel 1 beschrieben wurden.

Darüber hinaus bietet *Litera* die Möglichkeit, Bücher für eine Bestellung vorzumerken oder als Einzelbestellung in einen Warenkorb aufzunehmen. Diese Funktionen sind über das Menü „Bestellung“ erreichbar und werden in den Abschnitten 4.2 und 4.3 genauer beschrieben.

Zunächst sollen die einzelnen Maskenelemente vorgestellt werden.

4.1 Beschreibung der Maskenelemente

- **DokNr** die Dokumentnummer des angezeigten Dokuments; sie ist zusammen mit der Abteilung der Primärschlüssel in der zugrundeliegenden Datenbankrelation
- **Abteilung** Angabe, in welcher Abteilung (IfI A, B oder C) das Dokument zu finden ist
- **Erfassungsdatum** hier wird eingetragen, wann dieses Dokument in die Datenbank aufgenommen wurde
- **Autoren** Liste der Autoren des Dokuments
- **Titel** Angabe des Titels
- **Jahr** in welchem Jahr das Dokument veröffentlicht wurde
- **DokTyp** von welchem Typ das Dokument ist (*book*, *report*, *master thesis*, etc.)
- **Herausgegeben** falls die Autoren nur Herausgeber sind, wird das hier vermerkt
- **Sprache** Sprache, in der das Dokument verfasst wurde

- **Reihe** in welcher Zeitschriftenreihe das Dokument erschienen ist
- **Lieferant** wer das Dokument angeboten/geliefert hat
- **ISBN** ISBN des Dokuments
- **Auflage** Auflage des Dokuments
- **Betrag** Preis des Dokuments (nur Zahlenwert); die zugehörige Währung wird in der folgenden Auswahlliste namens
- **Währung** angegeben.
- **Verlag** in welchem Verlag das Dokument erschienen ist
- **Ort** Standort des Verlages
- **CR** CR-Klassifikation
- **Signatur** unter welcher Signatur das Dokument zu finden ist
- **Inventar-Nr** die Inventarnummer des Dokumentes
- **Interessent** Interessenten an dem Dokument, falls es bestellt werden soll
- **Status** Angabe über den Bestell- bzw. Inventarisierungszustand des Dokumentes
- **Ansicht** falls das Dokument bestellt werden soll, wird hier angegeben, ob dies nur zur Ansicht geschehen soll
- **Anmerkung** weitere Anmerkungen zum Dokument

4.2 Bestellungen

Litera kennt zwei unterschiedliche Arten der Bestellung: eine wird in der Datenbank gesammelt und zu einem vom Benutzer gewählten Zeitpunkt ausgeführt, die andere wird in einem Warenkorb gesammelt und muss vor dem Beenden des Programms vom Benutzer übernommen werden. Das zweite Verfahren wird im Abschnitt 4.3 beschrieben, das erstgenannte in diesem Abschnitt.

Soll ein Dokument in der Datenbank als „zu bestellen“ markiert werden, so geschieht dies, indem der Status auf „bestellen“ geändert wird.

Die so markierten Bücher können durch den Eintrag „Bestellung ausführen“ im Menü „Bestellung“ zu solch einer Bestellung zusammengestellt werden. Hierbei wird zunächst ein Fenster geöffnet, in dem die Verlage angegeben werden, bei denen Bücher zu bestellen sind. Zu jedem Verlag wird die Höhe der Bestellung in der eingestellten Rechnungswährung (die auch im Menü „Bestellung“ ausgewählt werden kann) angegeben.

In diesem Fenster kann der Benutzer nun den gewünschten Verlag auswählen, und mit dem Knopf „Bestellung vorbereiten“ ein weiteres Fenster öffnen, in dem die Details der in dieser Bestellung befindlichen Dokumente angezeigt werden. Die Daten aus diesem Fenster müssen vom Benutzer nun in den Bestellbrief übernommen werden, bevor er die Bestellung bestätigt.

Wird die Bestellung an dieser Stelle bestätigt, so werden die Dokumente als „bestellt“ markiert und mit dem Bestelldatum in eine Kontrolltabelle eingefügt.

Durch Auswahl des Menüeintrages „Kontrolltabelle“ kann die aktuelle Kontrolltabelle aus der Datenbank ausgelesen werden.

4.3 Einzelbestellungen und der Warenkorb

Litera bietet neben der Möglichkeit, Dokumente als „zu bestellen“ zu markieren, auch einen Warenkorb, mit dem Dokumente sofort bestellt werden können. Diese Bestellungen werden „Einzelbestellungen“ genannt.

Ein Dokument, das gerade angezeigt wird, kann durch den Menüeintrag „Einzelbestellung vormerken“ im Menü „Bestellung“ dem Warenkorb hinzugefügt werden, bzw. durch den Menüeintrag „Einzelbestellung löschen“ aus diesem wieder entfernt werden. Dadurch werden zunächst noch keine Änderungen an dem Datensatz vorgenommen.

Will man sich den Warenkorb anschauen, so geschieht dies mittels „Einzelbestellung selektieren“. Dadurch wird eine Anfrage an die Datenbank gestellt, die genau die Datensätze auswählt, die im Warenkorb liegen.

Soll die Einzelbestellung ausgeführt werden, so geschieht das über den Menüeintrag „Einzelbestellung ausführen“. Dadurch wird wie bei der normalen Bestellung zuerst ein Fenster geöffnet, in dem der Lieferant ausgewählt werden kann, und dann die entsprechenden Dokumente zu diesem Lieferanten in Listenform angezeigt. Die Daten in dieser Liste sollten in den Bestellbrief übernommen werden, und wenn die Bestellung bestätigt wird, werden die Dokumente in der Datenbank als „bestellt“ markiert und in die Kontrolltabelle eingetragen.

Teil II

Implementationsdetails

Kapitel 5

Die Programmstruktur

5.1 Überblick über die Struktur

Im zweiten Teil dieses Textes sollen die Verzeichnis- und Klassenstruktur des Programmpaketes erläutert werden.

Als erstes fällt auf, dass es unter den Klassen, die in der ursprünglichen Struktur von *Flit* enthalten sind, zwei verschiedene Typen gibt. Einerseits nämlich solche, die Metadaten der zugrundeliegenden Datenbankrelation beinhalten (wie z. B. Anzahl und Reihenfolge der Spalten, Datentypen, ob für einige Spalten NULL-Werte erlaubt sind etc.) oder die Beschreibung des Maskenlayouts. Diese Klassen müssen notwendigerweise für jede Maske individuell erzeugt werden – diese Arbeit übernimmt das Programm *MetaMask*. Darüber hinaus gibt es Klassen, die unverändert für alle Masken benutzt werden können, wie z. B. die graphischen Grundelemente. Diese zwei Typen von Klassen werden mit `Templates` bzw. `Shared` bezeichnet.

In den Templates wurden die relationsspezifischen Daten durch Labels im Programmtext ersetzt, die *MetaMask* mitteilen, wo die neuen Daten eingefügt werden müssen. Die genaue Beschreibung dieser Labels (und welche Daten an diesen Stellen eingesetzt werden) findet sich in Abschnitt 5.3.

Es sei an dieser Stelle gleich erwähnt, dass diese Labels auch noch in den individuell erzeugten Klassen existieren. Dadurch wird es *MetaMask* ermöglicht, auch auf bereits existierende Projekte zuzugreifen, und diese weiter zu überarbeiten; selbst dann, wenn sie schon mit zusätzlicher Funktionalität ausgestattet worden sein sollten.

Zusätzlich ist zu bemerken (wenn es auch nicht sonderlich relevant für die Praxis ist), dass auch *MetaMask* auf denselben Grundklassen aufbaut. Es ist dadurch möglich, auch die Maske des *MetaMask*-Hauptfensters mit *MetaMask* selbst zu bearbeiten.

Die Zusammenhänge zwischen den einzelnen Programmen bzw. den Klassengruppen werden in den folgenden Absätzen geschildert; hierzu sei auch auf Abbildung 5.1 hingewiesen; das „Tunneln“ durch *MetaMask* soll das Instantiieren

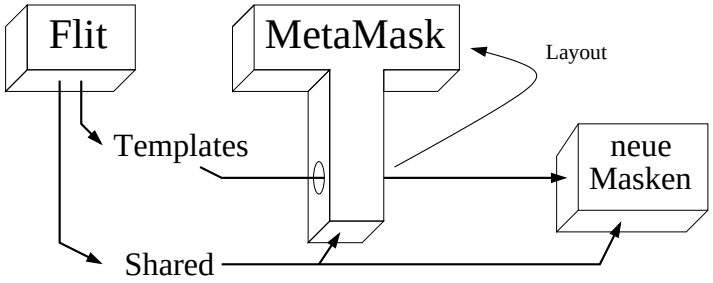

Abbildung 5.1: Übersicht der einzelnen Programmgruppen

der `Templates` zu den Klassen der neuen Maske veranschaulichen.

Die Zusammenhänge der zentralen Klassen werden in Abbildung 5.2 für *Flit* und Abbildung 5.3 für *MetaMask* dargestellt. Bei *Flit* sind dabei die Klassen, die relationsabhängige Daten enthalten, mit einem 'X' markiert, und solche, die zusätzlich für jede Maske erzeugt werden müssen, mit einem '+'.

Wie schon aus diesen beiden Abbildungen zu erkennen ist, sind die Zusammenhänge zwischen den einzelnen Klassen gering; sie werden vielmehr zentral im Arbeitsablauf zusammengeführt. Das ist insbesondere der Fall für die einzelnen separaten Fenster und ihre zugehörigen Listener-Interfaces, die in den Abbildungen nicht alle einzeln aufgeführt werden. Die einzige Klassengruppe, bei der etwas feinere Struktur vorhanden ist, ist die der Maskenelemente; diese Hierarchie ist gesondert in Abbildung 5.4 dargestellt. Es sei hier noch erwähnt, dass diese Klassen der graphischen Benutzeroberfläche sämtlich zu den `Shared`-Klassen gehören, so dass diese Struktur für alle erzeugten Masken (und *MetaMask*) gilt.

Damit die oben erwähnten `Template`- und `Shared`-Klassen von den Programmen gefunden werden können, befinden sie sich in einem Verzeichnis namens `dbis/mask/` bzw. Unterverzeichnissen davon. Die gemeinsamen Klassen finden sich in `dbis/mask/shared/`, und die (unvollständigen) Klassen, die für jede Maske individuell modifiziert werden müssen, liegen in `dbis/mask/templates/`; die erzeugten Klassen (für eine Maske namens „foo“) liegen dann in `dbis/mask/foo/`.

Als Wurzelverzeichnis muss im Loginfenster das Verzeichnis angegeben werden, in dem das im letzten Absatz erwähnte Verzeichnis `dbis/` zu finden ist; liegen die Klassen für *MetaMask* z. B. im Verzeichnis `/src/java/dbis/mask/metamask/`, so wäre als Wurzelverzeichnis `/src/java` anzugeben. Dies entspricht auch dem Verzeichnis, das im `CLASSPATH` auftreten muss.

Unterhalb dieser Struktur sind die Klassen auf jeweils drei weitere Verzeichnisse aufgeteilt: `ui/`, `net/` und `dok/`. Die Klassen werden dadurch wie folgt gruppiert:

- **ui/** für sämtliche graphischen Elemente; das umfasst neben den Maskenelemente auch die einzelnen Fenster und entsprechende Listener-Interfaces.
- **net/** für das Loginfenster und die Datenbankinformationen (Server und Port), sowie den Zugriff auf das ResourceBundle.

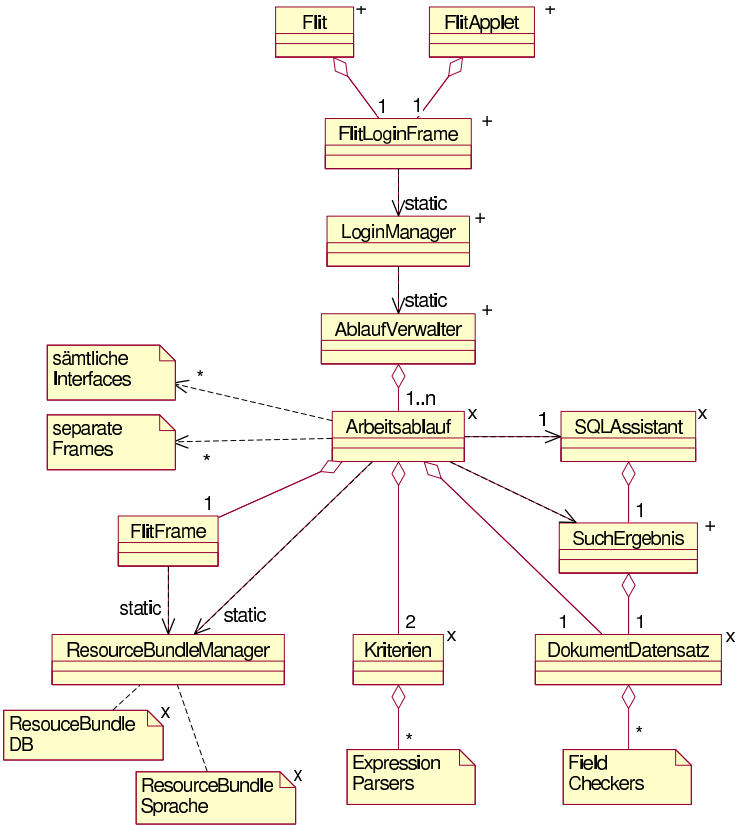

Abbildung 5.2: Modell der zentralen Klassen bei *Flit*

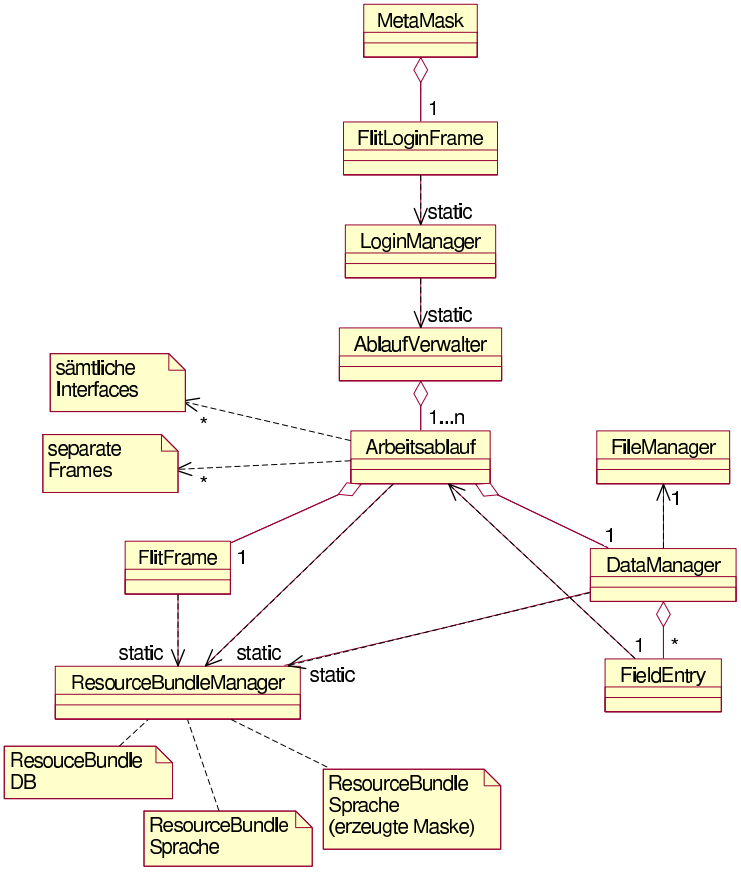

Abbildung 5.3: Modell der zentralen Klassen bei *MetaMask*

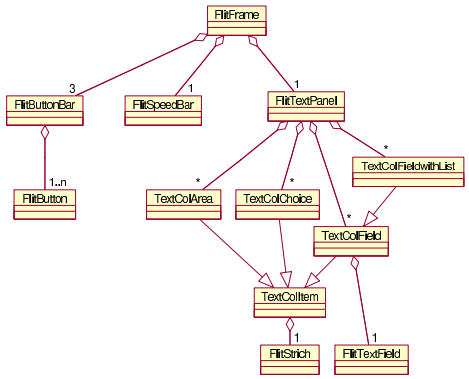

Abbildung 5.4: Modell der graphischen Elemente

- **dok/** für das Hauptprogramm, die SQL-Anfragen, die Daten enthaltenden Klassen und Eingabeprüfer.

In den nächsten zwei Abschnitten sollen die einzelnen Klassen erwähnt und ihre Funktion umrissen werden; genauere Dokumentation findet sich in Anhang 7.

5.2 Grundklassen: shared

5.2.1 Verzeichnis `dbis/mask/shared/ui/`

Interfaces

- FlitFocusListener wird aufgerufen, wenn der Fokus erhalten oder verloren wurde.
- TextColListener wird aufgerufen, wenn sich der Wert eines Textes geändert hat.
- CheckMenuListener wird aufgerufen, wenn ein CheckBoxMenuItem geändert wurde.
- CommandListener wird aufgerufen, wenn ein Befehl eingegeben wurde.

Die Maskenelemente

- FlitButton erweitert Button um eine Höhenangabe.
- FlitButtonBar sammelt FlitButtons in ein Panel.
- FlitStrich der FlitStrich gehört zu einem Maskenelement und zeigt dessen Status (modifiziert, fehlerhaft, deaktiviert) an.
- TextColItem diese abstrakte Klasse bildet die Grundklasse für alle Maskenelemente; sie enthält bereits einen FlitStrich und kann den Fokus annehmen.
- TextColField ist abgeleitet von TextColItem und stellt ein einfaches Textfeld dar.
- TextColFieldwithList Ist abgeleitet von TextColField und erweitert dessen Funktionalität um den Knopf, der das Listenfenster öffnet.
- TextColChoice ist abgeleitet von TextColField und stellt eine Auswahlliste dar .
- FlitSpeedBar die Anzeige mit Knöpfen für das Einlesen der Datensätze im Hintergrund.
- FlitTextField erweitert TextField in dem Sinne, dass es den Fokus nur akzeptiert, wenn es editierbar ist.

Panels

- FlitTextPanel das Hauptpanel, das die Maskenelemente enthält.

Dialoge & Fenster

- FlitErrorDialog stellt ein Fenster mit nicht-editierbarem Text und Buttons zum schließen zur Verfügung.
- FlitInfoFrame stellt ein Fenster mit nicht-focussierbarem Text und einem Button zum schließen zur Verfügung.
- FlitListFrame ein Fenster, das eine Liste aus einem Vector enthält.
- SaveFrame ein Fenster, das die Eingabe eines Filenames erwartet und einen Text speichert.
- ListendarstellungFrame ein Fester, das die Ausgabe des ListenAssistenten enthält.
- ExpertSQLEditor ein Fenster, das die Möglichkeit zum Editieren einer SQL-Anfrage bietet, mit entsprechenden Buttons.
- FlitFrame das Hauptfenster, das ein FlitTextPanel, eine FlitButtonBar, ein Menu, Statuszeilen und eine FlitSpeedBar enthält.
- ListAssistantFrame das Fenster des ListenAssistenten, verarbeitet die an es gegebenen Kommandos selbst.

Listener

- ListFrameListener wird aufgerufen, wenn ein Befehl in einem FlitListFrame ausgeführt werden soll.
- FlitKeyListener erweitert die Java-Klasse KeyListener um die Möglichkeit, Tasten auf Kommandos abzubilden, und gibt diese an einen CommandListener weiter.
- ListAssistantListener wird aufgerufen, wenn ein ListAssistantFrame ein Kommando erhält.

5.2.2 Verzeichnis `dbis/mask/shared/net/`

- RBManager der ResourceBundle-Manager bietet komfortable Möglichkeiten, auf Strings aber auch Vektoren von Strings in einem ResourceBundle zuzugreifen.

5.2.3 Verzeichnis `dbis/mask/shared/dok/`

Datenbankanbindung

- ConnectionManager verwaltet die Verbindungen an die Datenbank.

Suchergebnis

- SuchErgebnisCursor enthält eine Position in einem Suchergebnis und kann diese verschieben.

Feldtester für Werte

- FieldChecker Grundklasse für alle Feldtester; verwaltet den Fehler-String, erzeugt aber selbst noch keine Fehler.
- FieldCheckerString testet einen übergebenen String auf eine Maximallänge.
- FieldCheckerStringNE testet einen übergebenen String auf Maximallänge und „nicht leer“.
- FieldCheckerStringNQ testet einen übergebenen String auf Maximallänge und „nicht Fragezeichen“.
- FieldCheckerStringNEQ testet einen übergebenen String auf Maximallänge, „nicht leer“ und „nicht Fragezeichen“.
- FieldCheckerInt testet einen übergebenen String auf eine enthaltene Integerzahl mit Maximallänge.
- FieldCheckerIntNE testet einen übergebenen String auf eine enthaltene Integerzahl mit Maximallänge und „nicht leer“.
- FieldCheckerFloat testet einen übergebenen String auf eine enthaltene Kommazahl mit Vor- und Nachkommastellen.
- FieldCheckerFloatNE testet einen übergebenen String auf eine enthaltene Kommazahl mit Vor- und Nachkommastellen und auf „nicht leer“.
- FieldCheckerYear testet einen übergebenen String auf eine vierstellige Jahreszahl.

Feldtester für Ausdrücke

- ExprParser parst einen Ausdruck, der Klammern sowie die Zeichen „&“ (für „und“) und „|“ (für „oder“) enthalten darf, und übersetzt ihn in SQL.
- ExprStringParser parst einen Ausdruck und erzeugt eine entsprechende SQL-Ausgabe („LIKE '...' “), wobei auch „NULL“ beachtet wird.
- ExprIntParser parst einen Ausdruck mit Vergleichsoperatoren und Bereichsangaben der Form „a ... b“, wobei auch „NULL“ beachtet wird.

- ExprErfdatumParser parst einen Ausdruck mit Vergleichsoperatoren und Zeit-Einheiten.

5.3 Grundklassen: templates

In den Verzeichnissen der Templates liegen natürlich einerseits diejenigen Klassen, die Metadaten der speziellen Datenbankrelation enthalten sollen. Andererseits liegen dort auch solche Klassen, die auf Klassen der ersten Art zugreifen; da sie auf eine spezielle Instantiierung des Klassentemplates zugreifen müssen, können sie nicht in das Verzeichnis `shared` abgelegt werden. Es ist wahrscheinlich möglich, abstrakte Grundklassen für die Daten enthaltenden Klassen zu schaffen, was dieses Problem beseitigen würde, doch habe ich in der vorliegenden Fassung des Programmpaketes noch die ursprüngliche Struktur von *Flit* beibehalten.

In diesem Kapitel sollen daher zunächst sämtliche Klassen des Templateverzeichnisses erwähnt werden; genauere Erläuterungen, welche Daten in welchen Klassen enthalten sind, folgen im Abschnitt 5.4. Wie man die Klassen von Hand erweitern und modifizieren kann, wird in Kapitel 6 geschildert, und eine detaillierte Beschreibung der Klassen findet sich in Kapitel 7.

5.3.1 Verzeichnis `dbis/mask/templates/ui/`

- FlitLoginFrame stellt das Loginfenster und dessen Funktion bereit.
- `Sprache.properties` dieses ResourceBundle enthält sämtliche Daten für das Layout der Maske; darunter auch die Menü- und Knopfleisten, Hilfstexte etc..

5.3.2 Verzeichnis `dbis/mask/templates/net/`

- `DB.properties` dieses ResourceBundle enthält die Informationen, die zum Aufbau der Datenbankverbindung notwendig sind.
- LoginManager führt den Aufbau der Datenbankverbindung durch und übergibt die Programmkontrolle an den Ablaufverwalter.
- Flit enthält nur die Klasse `main`, die das Loginfenster öffnet.
- FlitApplet stellt eine Variante des Programms zur Verfügung, die als Applet gestartet werden kann.

5.3.3 Verzeichnis `dbis/mask/templates/dok/`

Klassen ohne relationsspezifische Daten

- BackgroundReader übernimmt das Einlesen der Datensätze für einen Arbeitsablauf im Hintergrund.

- SuchErgebnis bietet eine komfortable Schnittstelle zur Ergebnismenge einer Datenbankanfrage.
- ListManager erzeugt aus einem SuchErgebnis die Ausgabe in Listenform.
- SQLQuery erzeugt aus einem oder mehreren Kriterien eine SQL-Anfrage.
- AblaufVerwalter verwaltet das Öffnen und Schließen von (möglicherweise mehreren) Maskenfenstern.

relationsabhängige Klassen

- DokumentDatensatz enthält die einzelnen Daten eines Datensatzes.
- Kriterien enthält die Suchkriterien für eine Datenbankanfrage.
- SQLAssistant übernimmt die Kommunikation mit der Datenbank.
- Arbeitsablauf dies ist die zentrale Klasse der erzeugten Maske; sie steuert die Reaktion des Programms auf die meisten Eingaben im Hauptfenster.

5.4 die automatische erste Instantiierung

Die Templateklassen werden von *MetaMask* automatisch mit den Metadaten der Relation gefüllt und in das entsprechende Verzeichnis abgelegt. Die Grundfunktionalität der neuen Maske steht damit sofort zur Verfügung. In diesem Abschnitt wird erläutert, welche Daten das im einzelnen sind, und in welche Klassen sie geschrieben werden. Es ist jedoch nicht notwendig, diese Daten nachträglich von Hand zu modifizieren; um die Klassen konsistent zu halten, sollten derartige Veränderungen mittels *MetaMask* vorgenommen werden. Wie die erzeugte Maske um zusätzliche Funktionen erweitert werden kann, entnehme man Kapitel 6.

Damit *MetaMask* die Daten an die richtigen Stellen in den Templateklassen schreiben (und bei einem erneuten Aufruf auch von diesen Stellen lesen) kann, sind in diesen Klassen Kommentar-Paare eingefügt. Diese sind immer von der Form

```
// BEGIN labelname
// END labelname
```

Es ist nicht sinnvoll, zwischen die zwei Kommentare eines Paars Texte oder Daten einzufügen, da diese beim nächsten Speichern mit *MetaMask* überschrieben werden.

Die Stellen in den Klassen, an denen Daten und Funktionen eingefügt werden können, sind gekennzeichnet mit

```
// EDIT labelname
```

In den folgenden Abschnitten sollen die Klassen und die Daten vorgestellt werden, die von *MetaMask* bei den einzelnen Labels eingefügt werden.

5.4.1 Klasse DokumentDatensatz

Die Modifikationen im DokumentDatensatz werden von *MetaMask* vorgenommen; Veränderungen von Hand sind nicht zu empfehlen.

Der DokumentDatensatz enthält die Daten, aus denen ein einzelner Datensatz besteht. Dafür benötigt er Informationen, wie viele Spalten die Relation enthält, und wie diese heißen. Darüber hinaus versieht der DokumentDatensatz jedes Feld mit einem FieldChecker, um zu überprüfen, ob die eingegebenen Daten gültig sind; dafür werden Informationen zum Datentyp gebraucht. Letztendlich stellt der DokumentDatensatz auch einen Zugriff auf den Primärschlüssel des Datensatzes zur Verfügung.

DEF_LIST

Hier werden die Namen der Spalten in der Form

```
public static final String NAME = ''NAME'';
```

abgelegt und damit den Klassen sowohl in Form eines Strings (der Prozeduren übergeben werden kann) als auch in Form von Konstanten (die im Programmtext auftreten können) zur Verfügung gestellt. Es sei hier bemerkt, dass automatisch eine Konstante `ROWID` hinzugefügt wird.

ALL_ARRAY

Eine Aufzählung aller Spaltennamen-Konstanten.

SET_CHECKERS

An dieser Stelle in der Konstruktormethode werden den Feldern mit Hilfe der nachfolgenden privaten Methoden `add`XXX `(...)` die entsprechenden FieldChecker-Klassen (siehe Abschnitt 5.2.3) zugeordnet.

PRIMARY_KEY

In diesen Zeilen wird der Teil einer SQL-Anfrage in den `StringBuffer buf` geschrieben, der die Anfrage auf den Datensatz in der Datenbank einschränkt, dessen Primärschlüssel mit dem des aktuellen DokumentDatensatzes übereinstimmt – anders gesagt wird hiermit sichergestellt, dass wirklich nur dieser Datensatz in der Relation aktualisiert oder gelöscht etc. wird.

5.4.2 Klasse Kriterien

Die Modifikationen in den Kriterien werden von *MetaMask* vorgenommen; Veränderungen von Hand sind nicht zu empfehlen.

In dieser Klasse werden die Suchkriterien für eine Datenbankanfrage gespeichert; dafür werden Informationen über die Spalten der Relation benötigt. Außerdem wird jedes Feld mit einem ExprParser versehen, der die Angaben auf ihre Gültigkeit prüft und eine entsprechende SQL-Anfrage erstellen kann. Dafür muss selbstverständlich bekannt sein, um welchen Datentyp es sich in diesem Feld handelt.

Durch die Ähnlichkeit dieser Informationen zu denen im DokumentDatensatz kommt es zu etwas Redundanz in den Klassen, die allerdings so schon in *Flit* auftritt und deshalb noch nicht entfernt wurde.

DEF_LIST

Hier werden die Namen der Spalten in der Form

```
public static final String NAME = ''NAME'';
```

abgelegt. Es sei hier bemerkt, dass die Spalte `ROWID` nicht vokommt.

ALL_ARRAY

Eine Aufzählung aller Spaltennamen-Konstanten.

SET_PARSERS

An dieser Stelle in der Konstruktormethode werden den Feldern mit Hilfe der nachfolgenden privaten Methoden `add`XXX `(...)` die entsprechenden ExprParser-Klassen (siehe Abschnitt 5.2.3) zugeordnet.

5.4.3 Klasse SQLAssistant

Die Modifikationen im SQLAssistant werden fast alle von *MetaMask* vorgenommen; Veränderungen von Hand sind nur notwendig, wenn das Programm einige Werte selbständig ermitteln soll (Details hierzu finden sich in Kapitel 6).

Der SQL-Assistent übernimmt die Konstruktion von SQL-Anfragen. Dafür muss die Datenbankrelation spezifiziert werden, und außerdem müssen die Spalten in der Relation in der richtigen Reihenfolge angegeben werden, sowie diejenigen, die aktualisiert werden können, und welche als „NULL“ oder als programmgenerierter Wert eingefügt werden sollen. Darüber hinaus wird das Schreibrecht des Benutzers auf der Datenbankrelation überprüft.

NAMES

An dieser Stelle werden die Konstanten für den Besitzer und den Namen der Datenbankrelation abgelegt.

SORT_ARRAY

Hier sind die Namen der Spalten in der Relation (mit Bezug auf die öffentlichen Konstanten in der Klasse DokumentDatensatz) in der richtigen Reihenfolge abgelegt. Als erste Spalte wird hier automatisch `ROWID` angegeben.

NEW_ARRAY

Die Spalten, die ein neu erzeugter Datensatz umfasst, werden hier aufgeführt. Dabei ist zu beachten, dass es drei Formen von „magischen“ Strings gibt:

- der String beginnt mit '-': es soll immer der Wert „NULL“ eingefügt werden; dies ist dafür gedacht, wenn ein Datenbanktrigger den eigentlichen Wert erzeugen soll
- der String beginnt mit '+': die Maske soll einen neuen Wert erzeugen und diesen einfügen (siehe hierzu Kapitel 6)
- der String ist '.': signalisiert das Ende der Liste (dieser String wird automatisch als letztes eingetragen)

UPDATE_ARRAY

Bei dieser Marke werden die Spaltennamen eingetragen, die verändert werden dürfen; die Liste wird mit dem Wert „NULL“ abgeschlossen.

PERMISSION

Die Datenbankanfrage, die feststellt, ob der Benutzer Schreibrechte auf der Relation hat, wird an dieser Stelle zusammengestellt. Für das Verfahren der Schreibrechte in der *Flit*-Maskenfamilie siehe Abschnitt 5.5.

5.4.4 Klasse Arbeitsablauf

Der Arbeitsablauf ist die zentrale Klasse der Masken. Die grundlegenden Funktionen werden zwar schon von *MetaMask* eingetragen; wenn allerdings Knöpfe oder weitere Funktionen zur Maske hinzugefügt werden sollen, so muss diese Klasse von Hand verändert werden. Für kleinere Ergänzungen siehe Kapitel 6; sind jedoch größere Änderungen nötig, so muss diese Klasse eventuell in erheblichem Umfang überarbeitet werden.

CONST

Die Konstanten, die hier eingetragen werden, geben an, auf welche ResourceBundles zugegriffen werden soll; namentlich sind dies `Sprache.properties` im Verzeichnis `ui/` und `DB.properties` im Verzeichnis `net/` der Maske. Die beiden ResourceBundles werden in den folgenden Abschnitten genauer beschrieben.

LIST_COMMANDS

In diesen Abschnitt werden automatisch die Reaktionen auf die Knöpfe, die sich neben Textfeldern mit Listen befinden, eingetragen. Es sei nochmals darauf hingewiesen, dass die Reaktion des Programms auf weitere Knöpfe keinesfalls zwischen diesen beiden Marken eingetragen werden sollte, da sie sonst bei erneutem Speichern mit *MetaMask* überschrieben wird.

5.4.5 ResourceBundle `DB.properties`

Dieses sehr kurze ResourceBundle enthält die Informationen, die für die Datenbankverbindung nötig sind. *MetaMask* nimmt hier fast keine Änderungen vor; neue Werte sollten von Hand eingetragen werden.

DIRECT

In diesem Abschnitt liegen die Daten, die die Maske benutzt, wenn sie als selbständiges Programm ausgeführt wird. Dies sind namentlich `directURL` und `directDriver`, die Adresse der Datenbankinstanz und der JDBC-Driver.

REMOTE

Die Informationen für den Datenbankzugriff, wenn die Maske als Applet ausgeführt wird, werden in diesem Abschnitt angegeben. Entsprechend den oben genannten sind dies `remoteURL` und `remoteDriver`.

5.4.6 ResourceBundle `Sprache.properties`

Dieses ResourceBundle wird fast vollständig von *MetaMask* erzeugt; es kann dennoch mehrere Gründe geben, dieses File von Hand weiter zu verändern, denn die Daten, die dieses ResourceBundle enthält, sind vielfältig: sie umfassen Aussehen der Maske, Menüzeile, Hilfstexte, Werte für Auswahllisten etc. Die genauen Beschreibungen der automatisch generierten Werte werden in den folgenden Abschnitten aufgeführt; allerdings sollte für Veränderungen dieser Daten immer *MetaMask* benutzt werden, da sie sonst überschrieben werden könnten. Für Formate und Erläuterungen derjenigen Daten, die von Hand bearbeitet werden müssen, sei auf Kapitel 6 verwiesen.

VERSION

Hier fügt *MetaMask* einen Versionsstring ein, der das Datum des Speicherns enthält.

LAYOUT

An dieser Stelle wird das Aussehen der Maske festgelegt; Veränderungen sollten hier nur mit Hilfe von *MetaMask* vorgenommen werden, da dieser Bereich bei jedem Speichern neu geschrieben wird.

Sollen unbedingt Modifikationen vorgenommen werden, die von *MetaMask* vielleicht nicht unterstützt werden, so sei auf die Kommentarzeilen im File verwiesen, die diesen Marken vorangehen; es ist dann allerdings empfehlenswert, im erzeugten File diese Marken zu entfernen, damit die Änderungen nicht aus Versehen mit *MetaMask* überschrieben werden. Außerdem sei darauf hingewiesen, dass die eingetragenen Angaben vollständig und die Nummern fortlaufend sein müssen, da der RBManager bei fehlenden Resourcen das Layout als beendet erklärt.

ALWAYS_BLOCKED, SUCH_BLOCKED, NEU_BLOCKED, ERG_BLOCKED

Hier werden von *MetaMask* die Felder abgelegt, die in bestimmten Programmzuständen deaktiviert werden sollen.

CHOICES

Der Inhalt der Auswahllisten wird zwischen diesen Marken eingetragen. Für jede Liste wird dabei zur besseren Übersicht ein Trenner eingefügt, der auf das zugehörige Feld verweist.

LIST_SHORTHANDS

Die Namenskürzel, die die einzelnen Felder für die Listenausgabe bekommen haben, werden hier gespeichert. Diese Sektion wird von *MetaMask* bearbeitet – die verwandte Sektion `LIST_DEFAULTS` muss allerdings von Hand bearbeitet werden (siehe dazu Kapitel 6).

FOCUSHELP

Die Strings, die hier zu den einzelnen Feldern gespeichert sind, werden in den Statuszeilen des Hauptfensters angezeigt, wenn sich der Fokus im zugehörigen Feld befindet.

CURSOR_DOWN, CURSOR_UP

Ist die Option „Blättern mit Cursortasten“ ausgeschaltet, so bewegt sich der Cursor wie hier angegeben.

5.5 Die empfohlene Rollen-Verwaltung in Oracle

5.5.1 Schreibrechte

Die Masken der *Flit*-Familie haben eine spezielle Technik für Schreibrechte (INSERT/UPDATE/DELETE) auf den zugrundeliegenden Datenbankrelationen, die von *Flit* übernommen wurde.

Dazu wird in der Datenbank eine Sequenz mit dem Namen `foo_readwrite` für die Maske `foo` angelegt, auf die nur diejenigen Benutzer lesend zugreifen können, die auf der zur Maske gehörenden Relation Schreibrechte haben sollen. Loggt sich ein solcher Benutzer ein, so zählt die Sequenz um Eins weiter. Der SQL-Befehl, um eine solche Sequenz für eine Maske namens „foo“ anzulegen, ist der folgende:

```
create sequence foo_readwrite
    minvalue 1
    maxvalue 999999999999
    increment by 1
    start with 1
    cache 20
    order nocycle;
```

5.5.2 Die Rollen

Üblicherweise sollten für die verschiedenen Zugriffsarten auf die Datenbankrelationen, die mit den Masken der *Flit*-Familie möglich sind, Rollen angelegt werden, damit die entsprechenden Rechte einfach an Benutzer weitergegeben werden können. Es empfiehlt sich, zwischen drei Rollen zu unterscheiden:

- nur lesender Zugriff (Gast),
- lesender und schreibender Zugriff auf die Relation (berechtigter Benutzer) und
- Zugriff auf die Relation und Rechtevergabe an der Sequenz (Administrator).

Die SQL-Befehle, mit denen diese Rollen für eine Maske namens „foo“ auf einer Datenbankrelation namens „MAINTABLE“ eingerichtet werden, sind die folgenden:

```
create role foo_guest;
    grant select on MAINTABLE to foo_user;


create role foo_user;
    grant foo_guest to foo_user;
    grant insert, update, delete on MAINTABLE to foo_user;
    grant select on foo_readwrite to foo_user;


create role foo_admin;
    grant foo_user to foo_admin;
    grant all on foo_readwrite to foo_admin;
```

Diese Organisation der Rollen hat sich auch in dem Fall als nützlich erwiesen, wenn Werte bereits beim Einloggen abhängig vom Benutzer gesetzt werden sollen. Details hierüber können im Abschnitt 10.11 am Beispiel des Programms *Litera* nachgelesen werden.

Als Anmerkung sei hier noch hinzugefügt, dass Oracle es anscheinend verbietet, Rechte mit „grant option“ an Rollen zu vergeben. Daher muss der oben genannte komplette Zugriff auf die Sequenz namentlich jedem Administrator einzeln vom Besitzer der Sequenz gestattet werden, damit neue `foo_user` zugelassen werden können.

Kapitel 6

Wie werden die Masken erweitert?

Obwohl die von *MetaMask* erzeugten Masken schon mit der Grundfunktionalität für Suchen, Aktualisieren und Ergänzen der Datenbankrelation ausgestattet sind, kann es mehrere Gründe geben, die Files noch „von Hand“ weiter zu ergänzen.

Einerseits gibt es zwei Angaben, die von *MetaMask* noch nicht verwaltet werden: Die Sammlung der Listenformate für den Listenassistenten und das Menü mit der Auswahl der Sortierreihenfolge bei Datenbankanfragen. Wie diese im File `Sprache.properties` eingegeben werden können, beschreibt Abschnitt 6.1.1.

Soll die Grundfunktionalität einer Maske erweitert werden, sei es durch neue Knöpfe im Hauptpanel, neue Menüeinträge oder sogar Textfelder, die sich nicht auf die Hauptrelation beziehen, so muss das Hauptprogramm entsprechend erweitert werden. Die folgenden Abschnitte sollen einen Überblick darüber geben, was in diesen Fällen zu tun ist. Im Abschnitt 6.2.5 werden ein paar Fehlermeldungen erwähnt, die auftreten können, wenn Ergänzungen vom Programm nicht gefunden werden.

Generell sei darauf hingewiesen, dass die Angabe von Nummern immer von ’1’ beginnen und lückenlos fortlaufend sein muss, da der ResourceBundle-Manager bei fehlenden Einträgen die Eingabe als beendet erklärt.

6.1 Daten, die *MetaMask* nicht automatisch instantiiert

6.1.1 Listenformate

Die Formate, die der Listenassistent beim Starten anbietet, werden noch nicht von *MetaMask* verwaltet. Sie müssen daher separat in das ResourceBundle `Sprache.properties` eingetragen werden. Dies geschieht an der Stelle, die mit

```
// EDIT LIST_DEFAULTS
```

bezeichnet ist.

Einerseits können hier Listenformate angegeben werden, die im Listenassistenten durch drücken des Knopfes „Auswählen“ abgerufen werden können. Das Format dieser Eingaben muss das folgende sein:

```
list.format.X=STRING
```

wobei *X* eine fortlaufende Nummer ist, und *STRING* durch das gewünschte Listenformat ersetzt wird. Welche Bezeichnungen hier für die einzelnen Spalten gewählt werden können, wird im *MetaMask*-Hauptfenster durch den Eintrag „Kürzel für Listen“ festgelegt.

Außerdem zeigt der ListenAssistent eines dieser Listenformate schon beim Öffnen des Fensters an; in `Sprache.properties` wird auch eingestellt, welches dies ist, und zwar mittels eines Eintrags der Form

```
list.format.default=X
```

wobei *X* die Nummer des gewünschten Listenformates angibt.

6.1.2 das *OrderBy*-Menü

Die Auswahl, in welcher Ordnung das Suchergebnis einer Anfage ausgegeben werden soll, findet sich als Untermenü des Menüs „Anfrage“. Die möglichen Spalten, die als Sortierkriterien anwählbar sein sollen, können im Resource-Bundle `Sprache.properties` beim Label

```
// EDIT ORDER_BY
```

eingegeben werden. Die Form dieser Angaben ist

```
dok.menu.anfr.1.Y.ZZ=TEXT
```

wobei *Y* die laufende Nummer dieses Untermenüs ist. Für *ZZ* sind für jeden Eintrag drei Werte mit den entsprechenden Texten anzugeben. Dies sind

- **ty** Der Typ muss jeweils auf `c` gesetzt werden, damit dieser Menüeintrag als Checkbox erkannt wird.
- **ui** Dies ist der Text, der im Menü angezeigt wird; hier sollten deshalb die Spaltennamen mit benutzerfreundlichen Namen angegeben werden.
- **ac** Das *ActionCommand* muss mit dem String `ORDERBY_` beginnen, damit es der Arbeitsablauf als zu diesem Menü gehörig erkennt. Daran anschliessend müssen die Namen der Spalten angegeben werden, getrennt durch Kommata, aber ohne Leerzeichen. Diese Angabe ist diejenige, die in der SQL-Anfrage benutzt werden wird.

Zusätlich kann in der Angabe

```
dok.orderby.default=NR
```

eingestellt werden, welche dieser Reihenfolgen beim Programmstart aktiviert ist.

6.1.3 Werte der Datenbank-Anbindung

Die URL und der Driver, der von der erzeugten Maske benutzt werden soll, werden ebenfalls nicht automatisch von *MetaMask* eingetragen. Diese Werte sind im File `DB.properties` abgelegt. Es wird zwischen dem Aufruf als selbständiges Programm und dem als Applet unterschieden; für ersteres werden die Werte

`directURL=`*`TEXT`* und
`directDriver=`*`TEXT`*

verwendet, für letzteres die entsprechenden Werte

`remoteURL=`*`TEXT`* und
`remoteDriver=`*`TEXT`*

6.2 Ergänzungen des Hauptprogramms

6.2.1 Der Arbeitsablauf

Durch die erste Instantiierung werden aus den im Verzeichnis `template/` liegenden Files Klassenfiles hergestellt, die für jede Maske in einem gesonderten Verzeichnis abgelegt werden. Diese Files können im nachhinein modifiziert werden; wird *MetaMask* nochmals gestartet, so gehen die Ergänzungen nicht verloren. Dadurch werden das Erweitern der Klassen und das Umstellen des Layouts zwei voneinander unabhängige Vorgänge, deren Reihenfolge nicht festgelegt ist.

Wenn die Maske erweitert werden soll, so geschieht dies in der Klasse Arbeitsablauf. Deshalb sei hier kurz erwähnt, wie diese Klasse auf Eingaben des Benutzers reagiert (für eine genauere Beschreibung siehe Abschnitt 7.2).

Sämtliche Knöpfe und die meisten Menüeinträge geben ihre Signale in Form von *ActionCommands* mittels des Interfaces CommandListener an den Arbeitsablauf weiter. Die Reaktion auf diese Kommandos ist in vier Abschnitten der Methode `executeCommand (...)` festgelegt; von diesen Abschnitten beschäftigen sich drei mit Kommandos, die nur in einem bestimmten Programmzustand auftreten können, und der vierte mit den allgemeinen.

Die Reaktionen auf die Knöpfe in der Knopfleiste sowie der vorgegebenen Menüeinträge ist hier schon einprogrammiert und sollte nicht verändert werden. Die Knöpfe des Hauptpanels, die zu Textfeldern mit Listen gehören, werden im Programmabschnitt `LIST_COMMANDS` behandelt, der automatisch von *MetaMask* ergänzt wird. Darüber hinaus gehende Erweiterungen werden in den folgenden Abschnitten behandelt.

6.2.2 Hinzufügen von Knöpfen im Hauptpanel

MetaMask bietet die komfortable Möglichkeit, im Hauptpanel der erzeugten Maske neue Knöpfe einzufügen. Da deren Funktion aber nicht von *MetaMask* festgelegt werden kann, muss die Reaktion des Programms auf einen solchen Knopf nachträglich in die Klasse Arbeitsablauf eingetragen werden.

Das *ActionCommand*, das ein Knopf an den ARBEITSABLAUF weitergibt, entspricht dem Namen des Knopfes; dieser steht im allerersten Feld des *MetaMask*-Hauptpanels. Soll das Programm in jedem Programmzustand mit der selben Funktion auf den Knopf reagieren, so kann die Abfrage dieses Kommandos an der Stelle

```
// EDIT GENERAL_COMMANDS
```

eingefügt werden. Werden in verschiedenen Programmzuständen unterschiedliche Reaktionen gewünscht, so sollte der Code an den Stellen

```
// EDIT SUCH_COMMANDS,
// EDIT NEU_COMMANDS und
// EDIT ERG_COMMANDS
```

ergänzt werden. Hierbei ist allerdings darauf zu achten, dass Reaktionen für alle Zustände angegeben werden, in denen der Knopf aktiviert ist.

6.2.3 Hinzufügen von neuen Menüeinträgen

Das Erweitern der Menüs ist etwas aufwendiger als das Hinzufügen neuer Knöpfe, da *MetaMask* die Menüleiste nicht verwaltet. Um einen neuen Eintrag in ein Menü oder ein neues Menü einzufügen, muss das File `Sprache.properties` modifiziert werden.

Ein neues Menü wird in die Menüzeile eingetragen, indem an der Stelle

```
// EDIT MENU_BAR
```

die Liste

`dok.menu.`*`X`*`=`*`MENU`*

um die entsprechenden Einträge ergänzt wird. Hierbei ist *X* eine fortlaufende Zahl; es wird empfohlen, das Hilfsmenü als letzten Eintrag beizubehalten. Die Angabe *MENU* ist hierbei diejenige, die bei der Ausformulierung des Menüs als Kennzeichner benutzt wird.

Die einzelnen Menüs werden wie folgt angegeben: Jedes Menü hat einen Eintrag

`dok.menu.`*`MENU`*`.ui=`*`MENU_TITLE`*

der den Text darstellt, der in der Menüleiste sichtbar ist. Nachfolgend werden die einzelnen Einträge des Menüs ausformuliert, wobei das Format

`dok.menu.`*`MENU.X.ZZ=TEXT`*

vorgegeben ist. Hierbei ist *MENU* der Kennzeichner des Menüs und *X* die fortlaufende Nummer des Eintrages in diesem Menü. *ZZ* gibt den Typ des Wertes *TEXT* an, wobei die folgenden Möglichkeiten bestehen:

- **ui** sichtbarer Text des Eintrages; wird hierbei für *TEXT* der Wert '-' angegeben, so wird ein Separator eingefügt
- **ac** *ActionCommand*, das an den ARBEITSABLAUF weitergegeben wird, wenn dieser Menüeintrag ausgewählt wird

- **ty** Typ des Menüeintrages; für *TEXT* sind hierbei drei Einträge möglich:
  - **m** „Menüeintrag“ Dieser Wert braucht nicht angegeben zu werden, da er von den Masken als default-Wert gesetzt wird
  - **c** „*Checkbox*-Menüeintrag“ Erzeugt einen Menüeintrag, der an- und ausschaltbar ist. Die Behandlung dieser Einträge erfolgt im ARBEITSABLAUF gesondert und wird weiter unten genauer beschrieben.
    Es ist möglich, ein Untermenü mit *Checkbox*-Einträgen anzulegen. Hierbei ist es empfehlenswert, die *ActionCommands* eines *Checkbox*-Menüs mit einer charakteristischen Zeichenfolge zu beginnen, damit der ARBEITSABLAUF auf diese zusammengehörigen Eingaben erkennen kann.
  - **s** „*Submenu*“ Öffnet ein Untermenü, dessen Eintragungen in der Form

    `dok.menu.`*`MENU.X.Y.ZZ=TEXT`*

    angegeben werden. Dabei entspricht *X* dem obigen Wert, während *Y* eine fortlaufende Nummer dieses Untermenüs ist. Die Angaben *ZZ* und *TEXT* haben dieselben Wahlmöglichkeiten wie im Hauptmenü.
- **sc** „*ShortCut*“ Wird hier ein Buchstabe angegeben, so wird die entsprechende Kombination mit „CTRL“ zu einem Tastenkürzel. Diese Angabe kann einfach entfallen, wenn kein Tastenkürzel gewünscht ist.

In den meisten Fällen werden die *ActionCommands* der Menüeinträge mittels des COMMANDLISTENER-Interfaces an die Klasse ARBEITSABLAUF weitergegeben; die Behandlung dieser Kommandos entspricht genau der im Falle von Knöpfen und wurde im Abschnitt 6.2.2 beschrieben.

Für *Checkbox*-Menüeinträge ist dies etwas anders. Die Behandlung dieser Menüeinträge erfolgt über das Interface CHECKMENULISTENER und die Methode `checkMenuChanged (...)` im ARBEITSABLAUF. Dieser Methode wird das *ActionCommand* und ein Boolean-Wert übergeben, der anzeigt, ob der Menüeintrag an- oder ausgeschaltet ist.

Bei Untermenüs, bei denen einer von mehreren *Checkbox*-Einträgen ausgewählt werden soll, muss der ARBEITSABLAUF das Ausschalten des vorher ausgewählten Eintrags übernehmen. Als Beispiel für dieses Verfahren sei auf den Programmabschnitt des *OrderBy*-Menüs verwiesen, der durch

```
// EXAMPLE CHECKMENU
```

gekennzeichnet ist.

6.2.4 Hinzufügen von Textfeldern

Wie bei Knöpfen ist es auch bei Textfeldern möglich, diese mit Hilfe von *MetaMask* in das Layout einzufügen. Andererseits ist die Behandlung solcher Textfelder nicht ohne die Einarbeitung in die gesamte Programmstruktur möglich, da je nach Funktion dieser Textfelder neue Methoden, neue Klassen für Fenster, neue Listener oder ähnliches programmiert werden müssen.

Die Maskenelemente sind im Hauptfenster (Klasse FLITFRAME) gespeichert; für die Methoden dieser Klasse und die der Maskenelemente sei auf die Dokumentation in Kapitel 7 verwiesen.

6.2.5 Fehlermeldungen

Einige Fehlermeldungen können durch ungenaue Änderungen des ARBEITSABLAUFES entstehen. Dieser Abschnitt soll helfen, bei bestimmten Fehlermeldungen mögliche Ursachen zu finden.

- **Internes ActionCommand *AC* nicht erwartet** (Dieser Fehlermeldung wird der aktuelle Programmzustand vorangestellt) Ein *ActionCommand* wurde an die Methode `executeCommand (...)` übergeben, ohne dass der aktuelle Zustand dessen Behandlung vorsah. Dies kann geschehen, wenn die entsprechende Abfrage noch nicht oder nicht an der richtigen Stelle eingefügt wurde, oder wenn der Knopf nicht deaktiviert wurde. Diese Fehlermeldung tritt auch auf, wenn ein *Hotkey* gedrückt wurde, der im ARBEITSABLAUF nicht behandelt wird.
- **Internes ActionCommand *AC* im CheckMenuListener nicht erwartet.** Ein Checkbox-Menüeintrag wurde ausgewählt, dessen *ActionCommand* keiner Abfrage entsprach.
- Wenn eine **NullPointerException** in einer Programmzeile auftritt, in der ein TEXTCOLITEM benutzt werden soll, so wurde versucht, dieses aus dem Hauptfenster zu holen, und dabei wurde ein Name benutzt, zu dem kein Maskenelement existiert. Es kann sich bei dem gewünschten Element auch um einen Knopf handeln, denn diese werden getrennt von den anderen TEXTCOLITEMS behandelt: sie können direkt mit der Methode `setButtonEnabled (`*`NAME`*`, `*`ENABLED`*`)` des Hauptfensters aktiviert oder deaktiviert werden.

Kapitel 7

Dokumentation der einzelnen Klassen

In diesem Kapitel sollen die einzelnen Klassen mit ihren Methoden vorgestellt werden. Hierdurch soll jedoch weniger ein Überblick der Klassenstruktur gegeben werden; diese Funktion versucht Kapitel 5 zu erfüllen. Die folgenden Abschnitte sollen eher als Nachschlagewerk dienen und sind daher alphabetisch geordnet.

Die beiden ResourceBundles `DB.properties` und `Sprache.properties` werden hier nicht nochmals aufgeführt; für deren Beschreibung sei auf Abschnitt 5.4.5 für `DB.properties` bzw. Abschnitt 5.4.6 für `Sprache.properties` verwiesen.

7.1 Klasse AblaufVerwalter

extends java.lang.Object

Diese Klasse, die im Verzeichnis `dok/` instantiiert wird, beginnt und beendet Arbeitsabläufe und öffnet bzw. schließt die zugehörigen FlitFrames. Sie wird nur statisch benutzt.

Attribute:

-$aal: Vector

Ein Vektor, der alle aktiven Arbeitsabläufe enthält.

-$uff: Vector

Dieser Vektor enthält die unbenutzten FlitFrames.

-$MAX_UNUSED_FF: int

Die angegebene Konstante beschränkt die Anzahl der im Vektor *uff* gespeicherten unbenutzten FlitFrames.

Konstruktoren:

+AblaufVerwalter()

Default-Konstruktor; leer, da der AblaufVerwalter nur statisch benutzt wird.

Methoden:

+$setFont (font: Font): void

Setzt den Zeichensatz *font*, mit dem neue FlitFrames erzeugt werden sollen.

+$getOpenCount(): int

Ermittelt, wie viele Flit-Fenster gerade offen sind, und gibt diese Zahl zurück.

+$getArbeitsabläufe(): Enumeration

Liefert eine Aufzählung aller laufenden Arbeitsabläufe.

+$openNewWindow(): void

Erzeugt einen neuen Arbeitsablauf und öffnet ein neues Flit-Fenster. Tritt ein DB-Fehler beim Öffnen der neuen Verbindung auf, so wird eine SQLException geworfen.

+$openNewWindow (k: Kriterien)

Erzeugt einen neuen Arbeitsablauf und öffnet ein neues Flit-Fenster mit den gegebenen Kriterien *k*. Der Arbeitsablauf wird sofort einen DB-Anfrage mit diesen Kriterien stellen und das Ergebnis anzeigen. Tritt beim Öffnen der DB-Verbindung ein Fehler auf, so wird eine SQLException geworfen.

+$closing (al: Arbeitsablauf, ff: FlitFrame): void

Schließt den FlitFrame *ff* eines Arbeitsablaufes *al* und betrachtet diesen Arbeitsablauf ab jetzt als beendet.

+$close(): void

Veranlasst den AblaufVerwalter, alle gecachten unbenutzten FlitFrames zu vernichten.

-$getNewFlitFrame(): FlitFrame

Holt einen FlitFrame aus dem Vektor *uff* bzw. erzeugt einen neuen, falls *uff* leer ist.

-$keepMax (max: int): void

Entfernt vom Vektor *uff* die letzten FlitFrames, so dass nur *max* übrigbleiben.

7.2 Klasse ARBEITSABLAUF

extends java.lang.Object

implements CheckMenuListener, CommandListener, TextColListener, ListFrameListener, ListAssistantListener, FlitFocusListener

Diese Klasse ist die zentrale Klasse einer Maske und wird im Verzeichnis `dok/` instantiiert. Sie regelt den gesamten Arbeitsablauf für ein Flit-Fenster. Dazu gehören Veränderungen am Aussehen des Fensters (Menü- und Knopfleisten), aber auch Auslesen und Darstellen eines Datensatzes aus dem Suchergebnis und die Reaktion auf sämtliche Eingaben im Hauptfenster.

Attribute:

-$cm: CONNECTIONMANAGER

Der CONNECTIONMANAGER für die Verbindungen zur Datenbank.

-$gast: boolean

Gibt an, ob die Maske im Gast-Modus läuft.

-$inAnApplet: boolean

Gibt an, ob die Maske als Applet gestartet wurde.

-sqla: SQLASSISTANT

Der von diesem ARBEITSABLAUF benutzte SQLASSISTANT.

-br: BACKGROUNDREADER

Der BACKGROUNDREADER dieses ARBEITSABLAUF, der das Einlesen der Datensätze im Hintergrund ermöglicht.

-krit: KRITERIEN

Die aktuell benutzten, dargestellten, editierbaren KRITERIEN.

-autowildcards: boolean

Gibt an, ob die Felder, bei denen dies erlaubt ist, an beiden Enden automatisch mit Wildcards eingefasst werden sollen.

-casesensitive: boolean

Gibt an, ob die Suche auf Groß-Kleinschreibung achten soll.

-expert: boolean

Gibt an, ob der Benutzer in den Experten-Modus geschaltet hat.

-$nextprev_cursorupdown: boolean

Gibt an, ob der Benutzer mit den Cursor-Tasten zwischen Maskenfeldern (`True`) oder zwischen Datensätzen (`False`) wechseln will. Dieser Wert ist für alle Arbeitsabläufe gleich.

-dd: DOKUMENTDATENSATZ

Der aktuell benutzte, angezeigte (und möglicherweise editierbare) Datensatz.

-sucherg: SuchErgebnis

Das aktuell benutzte, möglicherweise editierbare Suchergebnis, aus dem der aktuelle Datensatz stammt.

-suchergcursor: SuchErgebnisCursor

Der Cursor des SuchErgebnisses *sucherg* an der aktuell dargestellten Position.

-ff: FlitFrame

Der von diesem Arbeitsablauf benutzte FlitFrame.

-state: int

Zustand des Arbeitsablaufes, entsprechend den nächsten drei Konstanten:

-$KRITERIENEINGABE: int

Arbeitsablauf befindet sich im Zustand „Kriterieneingabe“.

-$ERGEBNISDARSTELLUNG: int

Arbeitsablauf befindet sich im Zustand „Ergebnisdarstellung“.

-$NEUEINGABE: int

Arbeitsablauf befindet sich im Zustand „Neueingabe“.

-isModal: boolean

Wird auf `True` gesetzt, wenn die Maske gerade ein anderes Fenster anzeigt, das modal sein soll. Die Maske selbst akzeptiert keine Eingaben mehr, bis diese Variable wieder auf `False` gesetzt wird.

-$LABR*X*: int

Bezeichnet die Labelposition *X* (1...3) am rechten Fensterrand.

-$LABU*X*: int

Bezeichnet die Labelposition *X* (1...3) am unteren Fensterrand.

-$READAHEAD_OFF: int

Bezeichnet die Auswahl „Benutzer wünscht kein Vorauslesen“.

-$READAHEAD_SLOW: int

Bezeichnet die Auswahl „Benutzer wünscht langsames Vorauslesen“.

-$READAHEAD_FAST: int

Bezeichnet die Auswahl „Benutzer wünscht schnelles Vorauslesen“.

-readahead_speed: int

Gibt die momentane Auswahl der Vorauslesegeschwindigkeit an.

-displaying_speedbar: boolean

Gibt an, ob die Speedbar im Moment angezeigt wird.

Konstruktoren:

+Arbeitsablauf (ff: FLITFRAME)

Erzeugt einen neuen ARBEITSABLAUF im FLITFRAME *ff* mit entsprechender Initialisierung; der Startzustand ist „Kriterieneingabe“ und alle Textfelder sind leer.

+Arbeitsablauf (ff: FLITFRAME, k: KRITERIEN)

Erzeugt einen neuen ARBEITSABLAUF im FLITFRAME *ff* mit entsprechender Initialisierung; eine Datenbankanfrage mit den gegebenen KRITERIEN *k* wird sofort gestartet, und dann in den Zustand „Ergebnisdarstellung“ gewechselt.

Methoden:

+checkMenuChanged (ac: String, checked: boolean): void

Implementation des CHECKMENULISTENER-Interfaces.

+executeCommand (ac: String): void

Implementation des COMMANDLISTENER-Interfaces.

+executeListCommand (ac: String, FLITLISTFRAME flf, name: String, value: String): void

Implementation des LISTFRAMELISTENER-Interfaces.

+focusChanged (ac: String, have: boolean, temp: boolean): void

Implementation des FLITFOCUSLISTENER-Interfaces.

+listAssistant_executeCommand (ac: String): void

Implementation des LISTASSISTANTLISTENER-Interfaces.

+listAssistant_textColValueChanged (tci: TEXTCOLITEM, name: String, text: String): void

Implementation des LISTASSISTANTLISTENER-Interfaces.

+$setApplet (inAnApplet: boolean): void

Setzt die Kennzeichnung, ob die Maske als Applet ausgeführt wird, die für alle Arbeitsabläufe gemeinsam verwendet wird, auf *inAnApplet*.

+$setConnectionManager (cm: CONNECTIONMANAGER): void

Setzt den CONNECTIONMANAGER, der für alle Arbeitsabläufe gemeinsam verwendet wird, auf *cm*.

+$setGast (gast: boolean): void

Aktiviert bzw. deaktiviert den Gast-Modus, der für alle Arbeitsabläufe gemeinsam gilt, je nach dem, ob *gast* wahr oder falsch ist. Diese Methode muss vor dem ersten Aufruf eines Konstruktors dieser Klassen aufgerufen werden (bzw. erst wieder, nachdem alle vorher erzeugten Arbeitsabläufe nicht mehr in Gebrauch sind).

+textColValueChanged (tci: TEXTCOLITEM, name: String, text: String): void

Implementation des TEXTCOLLISTENER-Interfaces.

7.3 Klasse BackgroundReader

extends java.lang.Object

implements java.lang.Runnable

Der BackgroundReader, der im Verzeichnis `dok/` instantiiert wird, kann „im Hintergrund“ Daten eines Suchergebnisses vorab einlesen, damit diese später ohne Verzögerung zur Verfügung stehen und außerdem die Gesamtgröße des Suchergebnisses bekannt ist. Diese Klasse arbeitet stets sehr eng mit einem Arbeitsablauf zusammen.

Attribute:

-aa: Arbeitsablauf

Der Arbeitsablauf, zu dem dieser BackgroundReader gehört. In diesem wird der aktuelle Lese-Zustand dargestellt.

-se: SuchErgebnis

Das SuchErgebnis, in das die Datensätze eingelesen werden.

Konstruktoren:

+BackgroundReader (a: Arbeitsablauf)

Erzeugt einen neuen BackgroundReader zu dem übergebenen Arbeitsablauf *a*.

Methoden:

+isRunning(): boolean

Ermittelt, ob der BackgroundReader gerade als Thread läuft, und gibt in diesem Fall `True` zurück.

+pleaseStart (se: SuchErgebnis): void

Startet das Einlesen aller Dokumente in das Suchergebnis *se*. Falls der Thread bereits läuft, wird er zunächst gestoppt und dann neu gestartet.

+pleaseStop(): void

Stoppt das Einlesen. Diese Methode kehrt erst zurück, wenn der Einlese-Thread tatsächlich beendet ist. Falls er sich nach Aufforderung nach einer gewissen Zeitspanne nicht beendet hat, bricht ihn diese Methode gewaltsam ab. Falls der Thread nicht mehr läuft, ist der Aufruf wirkungslos.

+run(): void

Implementation des Runnable-Interfaces.

+setSlowSpeed (slow: boolean): void

Setzt die Lesegeschwindigkeit des BackgroundReaders. Diese Methode startet den Thread nicht! Der Default-Wert ist `True`, d. h. der BackgroundReader liest nur langsam ein; `False` setzt ihn auf maximale Lesegeschwindigkeit.

7.4 Interface CheckMenuListener

Der CheckMenuListener (Verzeichnis `shared/ui/`) ist ein Interface, das aufgerufen wird, wenn ein CheckBoxMenuItem in den Menüs geändert wurde.

Methoden:

+checkMenuChanged (ac: String, checked: boolean): void

Diese Methode wird aufgerufen, wenn ein CheckboxMenuItem geändert wurde. Der Arbeitsablauf, der dieses Interface implementiert, ändert entsprechende Voreinstellungen. Übergeben werden das *ActionCommand ac* des entsprechenden CheckboxMenuItems und die Angabe, ob es jetzt aktiviert ist (*checked* auf `True`).

7.5 Interface CommandListener

Der CommandListener (Verzeichnis `shared/ui/`) ist ein Interface, das aufgerufen wird, wenn der Benutzer im Hauptfenster einen Befehl gibt; dies ist bei fast jeder Aktion der Fall.

Methoden:

+executeCommand (ac: String): void

Wird aufgerufen, wenn ein Befehl ausgeführt werden soll.

7.6 Klasse ConnectionManager

extends java.lang.Object

Der ConnectionManager (Verzeichnis `shared/dok/`) verwaltet Connections an die Datenbank: Er öffnet neue Connections, wenn diese verlangt werden, und recycled benutzte Connections.

Konstruktoren:

+ConnectionManager (driver: String, url: String, username: String, password: String)

Erzeugt einen neuen ConnectionManager mit den entsprechenden Parametern: dem Driver *driver* für die Datenbank-Verbindung, der die Datenbank bei der gegebenen *url* erwartet, und den Benutzer *username* mit dem Passwort *password* einloggt.

Methoden:

+close(): void

Wird diese Methode aufgerufen, so schließt der ConnectionManager alle unbenutzten gecachten Connections.

+getNewConnection(): Connection

Gibt eine offene (neue oder wiederbenutzte) Verbindung an die Datenbank zurück. Diese ist auf `AutoCommit = False` gesetzt. Tritt ein Datenbank-Fehler auf, so wird eine SQLException geworfen.

+getUsername(): String

Gibt den in der Klasse gespeicherten Namen des Benutzers in der Datenbank zurück.

+recycle (con: Connection): void

Gibt einen benutzte Connection *con* an den ConnectionManager zurück. Es dürfen danach außerhalb des ConnectionManagers keinerlei Verweise auf diese Connection mehr existieren.

7.7 Klasse DokumentDatensatz

extends java.lang.Object

Ein DokumentDatensatz stellt einen Eintrag in der Datenbankrelation dar. Gleichzeitig werden hier die Veränderungen, die an diesem Datensatz vorgenommen werden, gespeichert, bevor sie in die Datenbank geschrieben werden; dies gilt auch für ein Löschen des Datensatzes. Ein DokumentDatensatz kann als „nicht modifizierbar“ markiert sein. Werden Veränderungen an diesem Datensatz vorgenommen, so wird er als exklusiv in der Datenbank gesperrt.

Jedes Feld wird durch einen String identifiziert. String-Konstanten für alle Feldnamen stehen statisch in dieser Klasse bereit, weshalb sie auch für jede neue Datenbankrelation erneut instantiiert werden muss; sie liegt daher im Verzeichnis `dok/`. Der Wert eines Feldes ist ebenfalls ein String, der allerdings nicht `Null` sein darf; zur Speicherung eines Datenbank-NULL-Wertes wird hier der leere String vereinbart.

Nicht aufgeführt werden hier die „public static final“ Konstanten, die die Namen der Spalten in der Datenbankrelation bezeichnen und erst durch das Instantieren in diese Klasse eingetragen werden; exemplarisch ist lediglich die immer vorhandene Konstante ROWID angegeben.

Attribute:

+$ROWID: String

Eine Konstante, um den Namen der Spalte ROWID der Datenbankrelation sowohl als Variable als auch als String zur Verfügung zu haben.

+$fieldnames: String[]

Eine Aufzählung der Namen aller Spalten der Datenbankrelation.

-orig: Hashtable

Der Hashtable mit den Originalwerten der Datensatz-Felder.

-modi: Hashtable

Der Hashtable mit Veränderungen an den Datensatz-Feldern. Sind für ein Feld keine Änderungen vorgenommen, so existiert der entsprechende Eintrag in diesem Hashtable nicht.

-modifiable: boolean

Gibt an, ob der DokumentDatensatz modifizierbar ist.

-exclusive: boolean

Gibt an, ob der DokumentDatensatz als exklusiv in der Datenbank gesperrt ist.

-deleted: boolean

Gibt an, ob der DokumentDatensatz vom Benutzer als gelöscht markiert worden ist.

Konstruktoren:

+DokumentDatensatz (modifiable: boolean)

Erzeugt einen neuen, leeren DokumentDatensatz mit entsprechenden FieldCheckern, der als modifizierbar markiert ist, wenn *modifiable* gleich `True` ist.

Methoden:

+check (feld: String): String

Überprüft den Inhalt des Feldes mit dem Namen *feld*; genaugenommen ruft diese Methode die private Methode `doCheck` auf.

#commit(): void

Übernimmt alle Modifikationen als neue Originalwerte. Der Datensatz ist damit (wieder) unmodifiziert.

+get (feld: String): String

Liefert den Wert des Feldes *feld* oder den leeren String, falls kein Wert gesetzt ist.

+getDat (feld: String): String

Liefert die ersten 16 Zeichen des Wertes des Feldes mit dem Namen *feld*, oder den leeren String, falls kein Wert gesetzt ist; dies wird benutzt zum Vergleich mit einem Datumsformat.

+getErrorString (feld: String): String

Holt das Ergebnis des letzten Check-Aufrufes für das Feld mit dem Namen *feld*. Ist kein Fehler aufgetreten, so wird `Null` zurückgegeben.

+getOriginal (feld: String): String

Holt den Originalwert des Feldes mit dem Namen *feld*.

+isDeleted(): boolean

Testet, ob der Datensatz als gelöscht markiert ist.

+isExclusive(): boolean

Testet, ob der Datensatz in der Datenbank zum exklusiven Zugriff gesperrt wurde.

+isModifiable(): boolean

Testet, ob der Datensatz modifizierbar ist.

+isModified(): boolean

Testet, ob der Datensatz modifiziert wurde.

+isModified (feld: String): boolean

Testet, ob das Feld mit dem Namen *feld* modifiziert wurde.

+restore(): void

Löscht alle Modifikationen und stellt den Originalzustand wieder her.

+restore (feld: String): void

Stellt den Originalzustand des Feldes mit dem Namen *feld* wieder her.

+set (feld: String, wert: String): String

Setzt den Wert des Feldes mit dem Namen *feld* auf *wert*. Das Feld ist damit modifiziert, falls der Wert ungleich dem gesetzten Originalwert ist. Falls für das Feld noch kein Originalwert gesetzt wurde, wird ein leerer String als Originalwert gesetzt. Der neue Wert wird von einem entsprechenden FieldChecker überprüft. Falls dieser einen Fehler findet, wird die Fehlermeldung als String zurückgegeben, sonst `Null`.
Ist der Datensatz nicht modifizierbar, so wird der Aufruf ignoriert; der Rückgabewert ist in diesem Fall ebenfalls `Null`.

#setDeleted (deleted: boolean): void

Markiert den Datensatz als gelöscht oder nicht gelöscht. Die Feldinhalte des Datensatzes bleiben in jedem Fall unverändert. Die Markierung ist nur dann möglich, wenn der Datensatz modifizierbar ist; ansonsten wird der Aufruf ignoriert.

#setExclusive (exclusive: boolean): void

Markiert den Datensatz als exklusiv oder nicht exklusiv.

#setModifiable (modifiable: boolean): void

Markiert den Datensatz als modifizierbar oder nicht modifizierbar. Falls der Datensatz als „nicht modifizierbar“ gekennzeichnet wird, werden sämtliche Modifikationen des Datensatzes verworfen und der Originalzustand wiederhergestellt.

#setOriginal (feld: String, wert: String): String

Setzt den Originalwert des Feldes mit dem Namen *feld* auf wert. Das Feld ist damit unmodifiziert. Der neue Wert wird von einem entsprechenden FieldChecker überprüft; falls dieser einen Fehler findet, wird die Fehlermeldung als String zurückgegeben; sonst `Null`.

+SQLprimaryKey (prefix: String): String

In dieser Methode wird beim Instantiieren eingetragen, welche Felder den Primärschlüssel der Datenbankrelation darstellen. Der Rückgabewert ist ein String mit der SQL-Anfrage, die in der Tabelle mit dem Namen *prefix* auf Gleichheit mit den Primärschlüsselwerten dieses DokumentDatensatzes testet.

-addXXX (name: String, [...]): void

Für jeden FieldChecker gibt es eine solche Methode, die eine neue Instanz des entsprechenden Checkers erzeugt und im DokumentDatensatz abspeichert.

-doCheck (feld: String, wert: String): String

Ruft den FieldChecker des Feldes mit dem Namen *feld* auf, falls dieser existiert, und lässt diesen den String *wert* überprüfen.

7.8 Klasse ExpertSQLEditor

extends java.lang.Dialog

implements ActionListener

Der ExpertSQLEditor (Verzeichnis `shared/ui/`) ist ein modaler Dialog der Maske. In Ihm kann der Benutzer, sofern er den Experten-Modus angeschaltet und „SQL-Anfrage editieren“ ausgewählt hat, die Anfrage modifizieren, bevor sie an die Datenbank abgeschickt wird.

Konstruktoren:

+ExpertSQLEditor (parent: Frame, sqlcmd: String)

erzeugt einen ExpertSQLEditor, der über dem *parent* Fenster erscheint und anfangs die Anfrage *sqlcmd* anzeigt. Der *parent* Frame wird blockiert, solange der Editor offen ist.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+getResult(): String

Liefert die evtl. modifizierte SQL-Anfrage, die sichtbar war, als der Dialog geschlossen wurde.

7.9 Klasse ExprErfdatumParser

extends ExprParser

Der ExprErfdatumParser (Verzeichnis `shared/dok/`) parst einen String; Items sind Dezimalzahlen mit nachfolgender Einheit ('d', 'D', 't' oder 'T' für

Tage; 'm' oder 'M' für Monate). Sie stellen die Anzahl der Tage bzw. Monate dar, die das Erfassungsdatum eines Dokuments zurückliegt.

Die folgenden weiteren Modifikatoren dürfen enthalten sein:

- <, <=, =, ==, >, >= Wird keine Relation vorangestellt, wird automatisch <= angenommen.
- Zahl1 ... Zahl2 (oder auch nur zwei Punkte)

Das gewünschte Dezimalzahlkriterium wird in der SQL-Where-Bedingung auf den Ausdruck `TRUNC(MONTHS_BETWEEN(SYSDATE,` *Spaltenname*`))` bei Monaten bzw. `TRUNC(MONTHS_BETWEEN(SYSDATE,` *Spaltenname*`) * 31 + 0.5)` bei Tagen angwendet.

Gültige Items sind z. B.:

- > 14 D – Dokumente, die vor mehr als 14 Tagen erfasst wurden
- 3..5 M – Dokumente, die vor 3 bis 5 Monaten erfasst wurden
- 2 D – Dokumente, die vor maximal 2 Tagen erfasst wurden

Aufrufer parsen einen String mit der Methode `parse` und holen sich, falls kein Fehler beim Parsen aufgetreten ist, danach den in SQL übersetzten String mit `getSQL`. Diese Methoden finden sich in der Klasse ExprParser.

Konstruktoren:

+ExprErfdatumParser()

Leerer Default-Konstruktor.

Methoden:

#parseItem(): boolean

Parst ein Item, das bei dieser Implementation von ExprParser eine Dezimalzahl mit Einheitsangabe ist, die mit gewissen „Verzierungen“ versehen sein darf. Es wird `True` zurückgegeben, falls das Parsen fehlerfrei durchgeführt werden konnte.

7.10 Klasse ExprIntParser

extends ExprParser

Der ExprIntParser (Verzeichnis `shared/dok/`) parst einen String; Items sind Dezimalzahlen.

Die folgenden weiteren Modifikatoren dürfen enthalten sein:

- <, <=, =, ==, >, >=
- Zahl1 ... Zahl2 (oder auch nur zwei Punkte)

Außerdem kann ein Item auch die Zeichenfolge „NULL“ oder das Prozentzeichen sein; die Semantik in SQL ist dann `IS NULL` bzw. `IS NOT NULL`.

Aufrufer parsen einen String mit der Methode `parse` und holen sich, falls kein Fehler beim Parsen aufgetreten ist, danach den in SQL übersetzten String mit `getSQL`. Diese Methoden finden sich in der Klasse ExprParser.

Konstruktoren:

+ExprIntParser()

Leerer Default-Konstruktor.

Methoden:

#parseItem()

Parst ein Item, das bei dieser Implementation von ExprParser eine Dezimalzahl ist, die mit gewissen „Verzierungen“ versehen sein darf. Es wird `True` zurückgegeben, falls das Parsen fehlerfrei durchgeführt werden konnte.

7.11 Klasse ExprParser

extends java.lang.Object

Der ExprParser (Verzeichnis `shared/dok/`) parst einen String. Diese abstrakte Klasse dient als Grundgerüst für verschiedene, spezialisierte Parser. Die (hier abstrakte) Methode `parseItem` muss bei abgeleiteten Klassen geeignet implementiert werden, um etwa eine Dezimalzahl oder ein Suchwort parsen zu können. Die abstrakte Klasse ExprParser stellt die gesamte Funktionalität für UND, ODER, NICHT und Klammerungen bereit.

Aufrufer parsen dann einen String mit der Methode `parse` und holen sich, falls kein Fehler beim Parsen aufgetreten ist, danach den in SQL übersetzten String mit `getSQL`.

Vor dem ersten Aufruf muss der SQL-Spaltenname mit `setSpalte` gesetzt werden.

Konstruktoren:

+ExprParser()

Leerer Default-Konstruktor.

Methoden:

+forbidAutoWildCards(): void

Verbietet das automatische Einkleiden des Suchwortes in WildCards für immer. Folgende Aufrufe von `setAutoWildCards` bleiben wirkungslos. Diese Methode dient dazu, gewisse Kriterienfelder dauerhaft ohne AutoWildCards benutzen zu können.

+getErrorPos(): int

Holt die vermeintliche Position des letzten Parse-Fehlers im zu parsen String. Die Position ist 0 <= pos <= s.length() im String s. Man beachte, dass == s.length() keine gültige Position darstellt; in diesem Fall war der Ausdruck vermutlich vorzeitig zu Ende. War kein Fehler aufgetreten, ist die Rückgabe undefiniert.

+getErrorString(): String

Holt die KlartextFehlermeldung des letzten Parse-Fehlers; war noch kein Fehler aufgetreten, wird `Null` zurückgegeben.

+getSQL(): String

Holt die SQL-Übersetzung des zuletzt geparsten Strings; ist beim Parsen ein Fehler aufgetreten, wird `Null` zurückgegeben.

+parse (s: String): boolean

Parst einen String. Es wird `True` zurückgegeben, wenn das Parsen fehlerfrei durchgeführt werden konnte.

#parseItem(): boolean

Parst ein Item. Die jeweilige Implementation dieser Klasse muss unbedingt sicherstellen, dass das Item auf jeden Fall durch die Zeichen '&', '|' und ')' terminiert wird. Dafür darf sie davon ausgehen, dass der zu parsende Teilstring keine führenden Leerzeichen mehr enthält. Der Rückgabewert sollte `True` genau dann sein, wenn das Parsen fehlerfrei durchgeführt werden konnte.

+setAutoWildCards (auto: boolean): void

Setzt das automatische Einkleiden des Suchwortes in WildCards. Default ist `False`, d. h. es ist deaktiviert.

+setCaseSensitive (casesensitive: boolean): void

Setzt die Case-Sensitivität des Suchwortes. Default ist `False`, d. h. Groß/Kleinschreibung wird nicht unterschieden.

+setSpalte (spalte: String): void

Setzt den SQL-Spaltennamen der Spalte, auf die sich dieser Parser beziehen soll.

#getChar(): char

Holt das aktuelle Zeichen des zu parsenden Strings. Liefert 0, falls keine Zeichen mehr da sind.

#nextChar(): void

Verbraucht das zuletzt gelesene Zeichen des zu parsenden Strings, d. h. die interne Cursor-Position wird um eins erhöht.

7.12 Klasse ExprStringParser

extends ExprParser

Der ExprStringParser (Verzeichnis `shared/dok/`) parst einen String; Items sind Strings.

Der String wird durch eines der Zeichen '&', '|' oder ')' beendet, wobei öffnende Klammern innerhalb des Strings selbstverständlich dazu führen, dass auch die zugehörige schließende Klammer mit in den String aufgenommen wird.

Die Semantik in SQL ist „`LIKE` string“, wobei Apostrophe angemessen behandelt werden. Führende und nachfolgende Leerzeichen werden abgeschnitten.

Das Item kann auch die Zeichenfolge „NULL“ sein; die Semantik in SQL ist dann „`IS NULL`“.

Aufrufer parsen einen String mit der Methode `parse` und holen sich, falls kein Fehler beim Parsen aufgetreten ist, danach den in SQL übersetzten String mit `getSQL`. Diese Methoden finden sich in der Klasse ExprParser.

Konstruktoren:

+ExprStringParser()

Leerer Default-Konstruktor.

Methoden:

#parseItem(): boolean

Parst ein Item, das bei dieser Implementation von ExprParser ein String ist. Der Rückgabewert ist `True`, wenn das Parsen fehlerfrei durchgeführt werden konnte.

+$quote (s: String): String

Ersetzt jeden in *s* vorkommenden Apostroph durch zwei Apostrophe.

7.13 Klasse FieldChecker

extends java.lang.Object

Der FieldChecker (Verzeichnis `shared/dok/`) überprüft einen String. Diese Klasse dient als Grundgerüst für verschiedene Checker. Die hier implementierte Methode `check` akzeptiert jeden beliebigen String. Sie muss bei abgeleiteten Klassen geeignet implementiert werden, um etwa eine Dezimalzahl prüfen zu können.

Aufrufer überprüfen einen String mit `check`. Falls ein Fehler beim Parsen aufgetreten ist, liefert das die Fehlermeldung als String. Der Checker merkt sich das Ergebnis der letzten Prüfung; es kann daher mit `getErrorString` immer wieder abgefragt werden.

Konstruktoren:

+FieldChecker()

Leerer Konstruktor.

Methoden:

+check (s: String): String

Überprüft einen String *s*, und gibt die Klartextfehlermeldung zurück, falls ein Fehler aufgetreten ist, ansonsten `Null`. Die Implementation dieser Methode in dieser Klasse akzeptiert sämtliche Strings.

+getErrorString(): String

Holt die Klartextfehlermeldung des letzten aufgetretenen Check-Fehlers, oder `Null`, wenn kein Fehler aufgetreten war.

7.14 Klasse FieldCheckerFloat

extends FieldChecker

Der FieldCheckerFloat (Verzeichnis `shared/dok/`) überprüft einen String, der eine Floatzahl enthalten soll, deren Stellenzahl eine gewisse maximale Stellenzahl vor und nach dem Punkt nicht überschreiten soll. Erlaubte Werte sind auch noch `Null` sowie der leere String.

Konstruktoren:

+FieldCheckerFloat()

Erzeugt einen neuen FieldCheckerFloat, der 8 Vor- und 2 Nachkommastellen akzeptiert.

+FieldCheckerFloat (max_vorkomma: int, max_nachkomma: int)

Erzeugt einen neuen FieldCheckerFloat, der *max_vorkomma* Vor- und *max_nachkomma* Nachkommastellen akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, der eine Floatzahl enthalten soll. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler aufrat.

7.15 Klasse FieldCheckerFloatNE

extends FieldCheckerFloat

Der FieldCheckerFloatNE (Verzeichnis `shared/dok/`) überprüft einen String, der eine Floatzahl enthalten soll, deren Stellenzahl eine gewisse Stellenzahl vor und nach dem Punkt nicht überschreiten soll. Nicht erlaubt sind `Null` und der leere String; in diesem Sinne steht der Name „NE“ für „not empty“.

Konstruktoren:

+FieldCheckerFloatNE (vorkomma: int, nachkomma: int)

Erzeugt einen neuen FieldCheckerFloatNE, der maximal *vorkomma* Vor- und *nachkomma* Nachkommastellen akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, der eine Floatzahl enthalten soll. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.16 Klasse FieldCheckerInt

extends FieldChecker

Der FieldCheckerInt (Verzeichnis `shared/dok/`) überprüft einen String, der eine Integerzahl enthalten soll, deren Stellenzahl eine gewisse maximale Stellenzahl nicht überschreiten soll. Erlaubte Werte sind auch noch `Null` sowie der leere String.

Konstruktoren:

+FieldCheckerInt()

Erzeugt einen neuen FieldCheckerInt, der maximal 8 Stellen akzeptiert.

+FieldCheckerInt (max: int)

Erzeugt einen neuen FieldCheckerInt, der maximal *max* Stellen akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, der eine Integerzahl enthalten soll. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.17 Klasse FieldCheckerIntNE

extends FieldCheckerInt

Der FieldCheckerIntNE (Verzeichnis `shared/dok/`) überprüft einen String, der eine Integerzahl enthalten soll, deren Stellenzahl eine gewisse maximale Stellenzahl nicht überschreiten soll. Nicht erlaubt sind hier `Null` und der leere String; in diesem Sinne steht der Name „NE“ für „not empty“.

Konstruktoren:

+FieldCheckerIntNE (max: int)

Erzeugt einen neuen FieldCheckerIntNE, der maximal *max* Stellen akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, der eine Integerzahl enthalten soll. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.18 Klasse FieldCheckerString

extends FieldChecker

Der FieldCheckerString (Verzeichnis `shared/dok/`) überprüft einen String, dessen Länge eine gewisse Maximallänge nicht überschreiten soll.

Konstruktoren:

+FieldCheckerString()

Erzeugt einen neuen FieldCheckerString, der Strings mit einer Maximallänge von 8 akzeptiert.

+FieldCheckerString (max: int)

Erzeugt einen neuen FieldCheckerString, der Strings mit einer Maximallänge von *max* akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, dessen Länge eine gewisse Maximallänge nicht überschreiten darf. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.19 Klasse FieldCheckerStringNE

extends FieldCheckerString

Der FieldCheckerStringNE (Verzeichnis `shared/dok/`) überprüft einen String, dessen Länge eine gewisse Maximallänge nicht überschreiten soll, und der weder `Null` noch der leere String sein darf; in diesem Sinne steht im Namen „NE“ für „not empty“.

Konstruktoren:

+FieldCheckerStringNE (max: int)

Erzeugt einen neuen FieldCheckerStringNE, der Strings mit einer Maximallänge von *max* akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, dessen Länge eine gewisse Maximallänge nicht überschreiten darf, und der weder der leere String noch `Null` sein darf. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.20 Klasse FieldCheckerStringNEQ

extends FieldCheckerString

Der FieldCheckerStringNEQ (Verzeichnis `shared/dok/`) überprüft einen String, dessen Länge eine gewisse Maximallänge nicht überschreiten soll, und der weder `Null` noch „?“ noch der leere String sein darf; in diesem Sinne steht im Namen „NEQ“ für „not empty nor questionmark“.

Konstruktoren:

+FieldCheckerStringNEQ (max: int)

Erzeugt einen neuen FieldCheckerStringNEQ, der Strings mit einer Maximallänge von *max* akzeptiert.

Methoden:

+check (s: String): String

Überprüft den String *s*, dessen Länge eine gewisse Maximallänge nicht überschreiten darf, und der weder der leere String noch `Null` noch „?“ sein darf. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.21 Klasse FieldCheckerStringNQ

extends FieldCheckerString

Der FieldCheckerStringNQ (Verzeichnis `shared/dok/`) überprüft einen String, dessen Länge eine gewisse Maximallänge nicht überschreiten soll, und der nicht „?“ sein darf; in diesem Sinne steht im Namen „NQ“ für „not questionmark“.

Konstruktoren:

+FieldCheckerStringNQ (max: int)

Erzeugt einen neuen FieldCheckerStringNQ, der Strings mit einer Maximallänge von *max* akzeptiert .

Methoden:

+check (s: String): String

Überprüft den String *s*, dessen Länge eine gewisse Maximallänge nicht überschreiten darf, und der nicht „?“ sein darf. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.22 Klasse FieldCheckerYear

extends FieldChecker

Der FieldCheckerYear (Verzeichnis `shared/dok/`) überprüft einen String, der eine Jahreszahl enthalten soll, deren Stellenzahl genau vier sein soll. Erlaubte Werte sind auch noch `Null` sowie der leere String.

Konstruktoren:

+FieldCheckerYear()

Leerer Konstruktor.

Methoden:

+check (s: String): String

Überprüft den String *s*, der eine vierstellige Jahreszahl enthalten soll. Rückgabewert ist die Klartextfehlermeldung, oder `Null`, falls kein Fehler auftrat.

7.23 Klasse Flit

extends java.lang.Object

Die Klasse Flit enthält lediglich die Methode `main` zum Starten der Maske; sie wird im Verzeichnis `net/` instantiiert. Prinzipiell kann man dieser Klasse für jede Maske einen entsprechenden neuen Namen geben; zur einfacheren Auszeichnung heißt sie jedoch für jede automatisch instantiierte Maske gleich. Eine Verwechslungsgefahr besteht nicht, da die Klassen in den zur Maske gehörigen Unterverzeichnissen liegen.

Konstruktoren:

+Flit()

Leerer Default-Konstruktor.

Methoden:

+$main(args: String[]): void

Erzeugt einen neuen FlitLoginFrame, der übergeben bekommt, dass das Programm nicht als Applet läuft. Die weitere Kontrolle obliegt lediglich diesem neuen Frame.

7.24 Klasse FlitApplet

extends java.applet.Applet

implements java.lang.Runnable

Für diese Klasse gelten dieselben Bemerkungen wie für die vorangegangene Klasse Flit, übertragen auf den Fall, dass die Maske als Applet gestartet wird. FlitApplet liegt ebenfalls im Verzeichnis `net/`.

Konstruktoren:

+FlitApplet()

Leerer Konstruktor; die eigentlich wichtigen Methoden sind `init` und `run`.

Methoden:

+init(): void

Initialisiert den RBManager derart, dass er weiß, von welchem Host er die ResourceBundles zu laden hat, und öffnet eine Java-Konsole.

+run(): void

Ruft die Methode `main` auf bzw. gibt einen StackTrace aus, falls dies zu einer Exception führt.

+start(): void

Beginnt einen neuen Thread.

+stop(): void

Beendet den laufenden Thread.

-main()

Öffnet einen neuen FlitLoginFrame, der übergeben bekommt, dass das Programm als Applet läuft. Die weitere Kontrolle obliegt dann lediglich diesem neuen Frame.

7.25 Klasse FlitButton

extends java.awt.Button

Ein FlitButton (Verzeichnis `shared/ui/`) ist ein Button, dem gleich beim Erzeugen ein *ActionCommand* übergeben werden muss, das dieser dann dem Arbeitsablauf übergibt, wenn er angeklickt wird. Außerdem kann man den Button auf Wunsch in der Höhe vergrößern, um eine größere Schaltfläche zu erhalten.

Konstruktoren:

+FlitButton (label: String, actionCommand: String, height: int)

Erzegt einen neuen FlitButton mit der Aufschrift *label* und der Höhe *height*, der den Befehl *actionCommand* weitergeben wird, wenn er angeklickt wird.

Methoden:

+getPreferredSize(): Dimension

Liefert die gewünschte Dimension dieses Buttons.

7.26 Klasse FlitButtonBar

extends java.awt.Panel

Die FlitButtonBar (Verzeichnis `shared/ui/`) ist ein Panel, das die drei Button-Leisten für die Modi „Kriterieneingabe“, „Ergebnisdarstellung“ und „Neueingabe“ der Maske enthält. Es ist nur jeweils eine dieser drei Button-Leisten sichtbar. Die Beschriftungen, die Höhe, die intern benutzten *Action-Commands* und die Reihenfolge der Buttons wird aus dem ResourceBundle `Sprache.properties` geladen.

Konstruktoren:

+FlitButtonBar (buttons: Hashtable, rb: ResourceBundle)

Erzeugt eine FlitButtonBar. Im Hashtable *buttons* werden die aus dem ResourceBundle *rb* gelesenen Buttons (mit ihrem Leistennamen+'.'+*ActionCommand* als Schlüssel) für späteren Zugriff gespeichert.

Methoden:

+displayBar (barname: String): void

Zeigt die gewünschte Button-Leiste mit dem Namen *barname* an. Die Namen der einzelnen Leisten wurden aus der Property-Datei gelesen.

7.27 Klasse FlitErrorDialog

extends java.awt.Dialog

implements ActionListener

Der FlitErrorDialog (Verzeichnis `shared/ui/`) ist ein modaler Dialog der Maske, der einen (auch mehrzeiligen) Text anzeigt.

Konstruktoren:

+FlitErrorDialog (parent: Frame, font: Font, title: String, infotext: String, buttontext: String)

Erzeugt einen FlitErrorDialog mit dem Titel *title*, der einen (auch längeren, mehrzeiligen) Text *infotext* und einen Button zum Schließen des Dialogs anzeigt, der die Bezeichnung *buttontext* trägt. Der Frame *parent* ist blockiert, so lange der Dialog geöffnet ist. Das Dialogfenster wird so auf dem Bildschirm positioniert, dass es zentriert über dem *parent* Frame erscheint, ohne sich jedoch über den Bildschirmrand hinaus zu erstrecken.

+FlitErrorDialog (parent: Frame, font: Font, title: String, infotext: String, buttontexte: String[], defbutton: int)

Dieser Konstruktor unterscheidet sich von dem vorangegangenen nur dadurch, dass der erzeugte FlitErrorDialog mehrere Buttons enthält, die zum Schließen benutzt werden können. Die Beschriftungen der Buttons werden (von links nach rechts) in *buttontexte* übergeben, der Default-Button kann mittels *defbutton* angegeben werden, wobei der erste Button die Nummer 0 trägt.

Methoden:

+actionPerformed (a: ActionEvent): void

Implementation des ActionListener-Interfaces.

+getResult(): int

Liefert die Nummer des Buttons, mit dem der Dialog geschlossen wurde, wobei der erste Button die Nummer 0 trägt.

7.28 Interface FlitFocusListener

Der FlitFocusListener (Verzeichnis `shared/ui/`) ist ein abstraktes Interface, das aufgerufen wird, wenn der Focus erhalten oder verloren wurde.

Methoden:

+focusChanged (ac: String, have: boolean, temp: boolean): void

Wird aufgerufen, wenn der Focus erhalten oder verloren wurde. Dabei ist *ac* das *ActionCommand* der aufrufenden Komponente. *have* ist `True`, wenn der Focus erhalten wurde, und *temp* ist `True`, falls die Komponente den Focus nur temporär verloren hat, andernfalls `False`.

7.29 Klasse FlitFrame

extends java.awt.Frame

implements ItemListener, ActionListener

Der FlitFrame (Verzeichnis `shared/ui/`) ist das Hauptfenster der Masken. Es enthält ein FlitTextPanel, in dem die Datensätze dargestellt werden, eine FlitButtonBar am rechten Fensterrand, ein Menü, Statuslabel am unteren Fensterrand und eine FlitSpeedBar in der rechten unteren Ecke. Die Anordnung und Beschriftung des Menüs wird bei den erzeugten Masken aus dem ResourceBundle Sprache.properties geladen, es ist jedoch auch möglich, andere Namen anzugeben. Die FlitSpeedBar lässt sich auch ausblenden.

Attribute:

-acWinClose: String

Das *ActionCommand*, das an den CommandListener weitergegeben wird, falls das Fenster geschlossen wird.

-withSpeedBar: boolean

Gibt an, ob das Fenster überhaupt eine SpeedBar enthalten soll. Das ist nicht gewünscht, wenn diese Klasse für *MetaMask* benutzt wird.

-panelMain: Panel

Das Hauptpanel im FlitFrame.

Konstruktoren:

+FlitFrame (font: Font, RBName: String, withSpeedBar: boolean)

Erzeugt einen FlitFrame, der vom Aufrufer mit `setVisible(True)` angezeigt werden muss. Er benutzt den Font *font* und lädt die Informationen über die graphischen Elemente aus dem ResourceBundle *RBName*. Der FlitFrame versucht nur dann, eine SpeedBar zu erzeugen, wenn *withSpeedBar* auf `True` gesetzt ist; das ist allerdings nur bei *MetaMask* nicht gewünscht.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+beep(): void

Gibt einen Piep aus.

+detachListeners(): void

Entfernt alle Referenzen auf Listener (TextColListener, FlitFocusListener, CheckMenuListener, CommandListener) aus diesem FlitFrame und den enthaltenen Komponenten.

+displayBar (barname: String): void

Zeigt die Button-Leiste mit dem Namen *barname* an; die möglichen Namen sind aus der Property-Datei geladen worden.

+displaySpeedBar (barname: String): void

Setzt die FLITSPEEDBAR in den gewünschten Anzeigezustand mit dem Namen *barname*.

+findTextColItem (name: String): TEXTCOLITEM

Liefert das TEXTCOLITEM mit dem Namen *name*. Falls kein solches existiert, wird `Null` zurückgegeben.

+getButton (ac: String): Button

Liefert den Button mit dem *ActionCommand ac*.

+getKeyListener(): KeyListener

Liefert den von diesem FLITFRAME verwendeten KeyListener.

+getTextColChoices(): Enumeration

Liefert eine Enumeration aller TEXTCOLCHOICES.

+itemStateChanged (e: ItemEvent): void

Implementation des ItemListener-Interfaces.

+replacePanel (rbName: String): void

Löscht das Hauptpanel und erschafft ein neues, das aus dem Resource-Bundle *rbName* gelesen wird. Dies wird für die Testframes bei *MetaMask* benutzt.

+requestFocusInTextColItem (name: String): void

Setzt den Input-Focus in das TEXTCOLITEM mit dem Namen *name*. Falls keines mit diesem Namen existiert oder der Wert `Null` übergeben wird, ist der Aufruf wirkungslos.

+setButtonEnabled (ac: String, enabled: boolean): void

Aktiviert oder deaktiviert den Button mit dem *ActionCommand ac* je nach dem Wert von *enabled*. Es ist egal, ob sich dieser Button in der FLITBUTTONBAR am rechten Fensterrand oder im FLITTEXTPANEL befindet.

+setCheckMenuEnabled (ac: String, enabled: boolean): void

Aktiviert oder deaktiviert das CheckboxMenuItem mit dem *ActionCommand ac* je nach dem Wert von *enabled*.

+setCheckMenuListener (cml: CHECKMENULISTENER): void

Setzt einen CHECKMENULISTENER für diesen FLITFRAME. Es ist nur ein einziger CHECKMENULISTENER pro FLITFRAME vorgesehen, d. h. nur der zuletzt gesetzte CHECKMENULISTENER wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein CHECKMENULISTENER mehr registriert.

+setCheckMenuState (ac: String, checked: boolean): void

Setzt den Zustand des CheckMenuItems mit dem *ActionCommand ac*. Das CheckMenuItem wird angeklickt dargestellt, wenn *checked* auf `True` gesetzt ist.

+setCommandListener (cl: CommandListener): void

Setzt einen CommandListener für diesen FlitFrame. Es ist nur ein einziger CommandListener pro FlitFrame vorgesehen, d. h. nur der zuletzt gesetzte CommandListener wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein CommandListener mehr registriert.
Der CommandListener wird über die folgenden Benutzerbefehle informiert:

- betätigen von Buttons
- auswählen von Menüpunkten
- drücken von HotKeys
- Versuch des Schließens des Fensters über den Fenstermanager

+setFirstFocus (name: String): void

Setzt das TextColField, das beim ersten Öffnen des Fensters sofort den Focus haben soll.

+setFlitFocusListener (ffl: FlitFocusListener): void

Setzt einen FlitFocusListener für diesen FlitFrame. Er wird in allen in diesem FlitFrame enthaltenen TextColItems gesetzt. Es ist nur ein einziger FlitFocusListener pro FlitFrame vorgesehen, d. h. nur der zuletzt gesetzte FlitFocusListener wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein FlitFocusListener mehr registiert.

+setHotkeyMapping (h: Hashtable): void

Setzt eine Übersetzungstabelle für die *Hotkeys* dieses FlitFrames. Darf mit `Null` aufgerufen werden; in diesem Fall wird dann keine Übersetzungstabelle genutzt.
Die Tabelle *h* kann die Strings „F1“ bis „F12“ sowie „PAGE_UP“ und „PAGE_DOWN“ als Schlüssel enthalten und diesen Tasten andere *ActionCommands* zuordnen. Beim Drücken eines *Hotkeys* wird der registrierte CommandListener mit den übersetzten *ActionCommands* aufgerufen. Falls zu einer Taste kein Schlüssel in der Übersetzungstabelle existiert, wird der CommandListener mit dem originalen *ActionCommand* aufgerufen.

+setLabel (n: int, text: String): void

Setzt den Text des *n*ten Labels. Die Labels werden von 0 an durchgezählt, zunächst von oben nach unten die Labels unter der Buttonleiste, danach von oben nach unten die Labels unter der Eingabemaske. Es stehen in der Klasse Arbeitsablauf die Konstanten „LABRx“ bzw. „LABUx“ zur Verfügung, wobei x Werte von 1 bis 3 annehmen kann.

+setMenuEnabled (name: String, enabled: boolean): void

Aktiviert oder deaktiviert das Menü mit dem Namen *name* je nach dem Wert von *enabled*.

+setMenuItemEnabled (ac: String, enabled: boolean): void

Aktiviert oder deaktiviert das MenuItem mit dem *ActionCommand ac* je nach dem Wert von *enabled*.

+setTextColListener (tcl: TEXTCOLLISTENER): void

Setzt einen TEXTCOLLISTENER für diesen FLITFRAME. Er wird in allen in diesem FLITFRAME enthaltenen TEXTCOLITEMS gesetzt. Es ist nur ein einziger TEXTCOLLISTENER pro FLITFRAME vorgesehen, d. h. nur der zuletzt gesetzte TEXTCOLLISTENER wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein TEXTCOLLISTENER mehr registriert.

+setVisibleAbbruch (visible: boolean): void

Setzt den Button „Abbruch“ der SpeedBar auf sichtbar oder unsichtbar je nach Wert von *visible*.

-fillMenu (rb: ResourceBundle, prefix: String, menu: Menu): void

Liest die Einträge eines Menüs aus dem ResourceBundle *rb*, wo es durch *prefix* gekennzeichnet ist. Die gelesenen Informationen und Submenüs werden in das Menu *menu* eingebaut.

7.30 Klasse FLITINFOFRAME

extends java.awt.Frame

implements ActionListener

Der FLITINFOFRAME (Verzeichnis `shared/ui/`) ist ein selbstständiges Informationsfenster der Masken. Er enthält einen (auch mehrzeiligen) Text und einen Button zum Schließen des Fensters.

Konstruktoren:

+FlitInfoFrame (parent: Frame, font: Font, title: String, infotext: String, buttontext: String)

Erzeugt einen neuen FLITINFOFRAME über dem Frame *parent* mit dem Titel *title*, der den (auch längeren, mehrzeiligen) Text *infotext* enthält. Der Button zum Schließen des Fensters ist mit *buttontext* beschriftet.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

7.31 Klasse FlitKeyListener

extends java.awt.Object

implements KeyListener

Der FlitKeyListener (Verzeichnis `shared/ui/`) ist der KeyListener der Maske. Er reagiert auf *Hotkey*-Tastendrücke und benachrichtigt dann einen CommandListener.

Die *Hotkeys* und die zugehörigen *ActionCommands* sind aus technischen Gründen in den Sourcecode dieser Klasse fest eingebaut. Die Tasten „page up“ und „page down“ liefern die *ActionCommands* „PAGE_UP“ bzw. „PAGE_DOWN“, die Tasten „cursor up“ und „cursor down“ liefern die *ActionCommands* „CURSOR_UP“ bzw. „CURSOR_DOWN“, die Funktionstasten F1 bis F12 die *ActionCommands* „F1“ bis „F12“. Die *ActionCommands* lassen sich mit Hilfe einer Übersetzungstabelle, die mit der Methode `setMapping` eingetragen wird, ändern.

Der FlitKeyListener „schluckt“ die KeyEvents, die die oben genannten Tasten liefern, d. h. er ruft die Methode `consume` der Events auf.

Attribute:

+keys: String[]

Ein Array mit allen „originalen“ *ActionCommand*-Strings zu den *Hotkeys*.

Konstruktoren:

+FlitKeyListener()

Erzeugt einen neuen FlitKeyListener und initialisiert ihn mit den in der Beschreibung dieser Klasse erwähnten *ActionCommands* für die entsprechenden KeyEvents.

Methoden:

+keyPressed (e: KeyEvent): void

Nimmt ein KeyEvent auf und übersetzt es in das entsprechende *ActionCommand*, das dann an den registrierten CommandListener weitergegeben wird.

+keyReleased (e: KeyEvent): void

Zur Implementation des KeyListener-Interfaces; leer.

+keyTyped (e: KeyEvent): void

Zur Implementation des KeyListener-Interfaces; leer.

+setCommandListener (cl: CommandListener): void

Setzt einen CommandListener für diesen FlitKeyListener. Es ist nur ein einziger CommandListener pro FlitKeyListener vorgesehen, d. h. nur der zuletzt registrierte CommandListener wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein CommandListener mehr registriert.

+setMapping (h: Hashtable): void

Setzt eine Übersetzungstabelle für die *Hotkeys* diese FlitKeyListeners. Darf mit `Null` aufgerufen werden; in diesem Fall wird dann keine Übersetzungstabelle benutzt.
Die Tabelle *h* kann die Strings „F1“ bis „F12“ sowie „PAGE_UP“ und „PAGE_DOWN“ als Schlüssel enthalten und diesen Tasten andere *ActionCommands* zuordnen. Beim Drücken eines *Hotkeys* wird der registrierte CommandListener mit den übersetzten *ActionCommands* aufgerufen. Falls zu einer Taste kein Schlüssel in der Übersetzungstabelle existiert, wird der CommandListener mit dem originalen *ActionCommand* aufgerufen.

-add (keycode: int, ac: String): void

Diese Methode weist dem numerischen KeyEvent *keycode* das „originale“ *ActionCommand ac* zu.

-close(): void

Schließt die Aufnahme von KeyEvents ab.

7.32 Klasse FlitListFrame

extends java.awt.Frame

implements KeyListener, WindowListener, ActionListener

Der FlitListFrame (Verzeichnis `shared/ui/`) ist ein Fenster der Masken, das eine Auswahlliste enthält. Der Inhalt der Liste wird durch einen Vektor von Strings vorgegeben.

Konstruktoren:

+FlitListFrame (v: Vector, lfl: ListFrameListener, name: String, font: Font)

Erzeugt einen neuen FlitListFrame aus einer im Vektor *v* gegebenen Liste von Strings. Es muss ein ListFrameListener *lfl* angegeben werden, der über die auftretenden Events benachrichtigt wird. Der Name *name* dient ausschließlich dem ListFrameListener zur Identifikation dieses Frames.

Methoden:

+actionPerformed (a: ActionEvent): void

Implementation des ActionListener-Interfaces; gibt das *ActionCommand* zum ActionEvent *a* an den ListFrameListener weiter.

+getPreferredSize(): Dimension

Holt die bevorzugte Größe des Frames. Ist diese schmaler als die gewünschte Breite (mit `pleaseBeAsWideAs` gesetzt), so wird der Wert des Benutzers zurückgegeben.

+keyPressed (e: KeyEvent): void

Schließt den Frame, wenn Escape gedrückt wird.

+keyReleased (e: KeyEvent): void

Implementation des KeyListener-Interfaces; leer.

+keyTyped (e: KeyEvent): void

Implementation des KeyListener-Interfaces; leer.

+pleaseBeAsWideAs (w: int): void

Schlägt dem FlitListFrame eine gewünschte Breite in Pixeln vor.

+pleaseDispose(): void

Veranlasst den FlitListFrame, sich zu schließen und alle Resourcen freizugeben. Darf mehrfach aufgerufen werden – alle Aufrufe nach dem ersten sind dann wirkungslos.

+pleasePositionHere (p: Point): void

Positioniert den FlitListFrame auf dem Bildschirm. Die übergebene Position *p* wird dabei nur als Empfehlung betrachtet, von der abgewichen wird, falls der FlitListFrame sonst nicht mehr vollständig auf dem Bildschirm sichtbar wäre.

+pleasePositionSouthWestOfThisWithMinWidth (com: Component, tcf: TextColField): void

Positioniert den FlitListFrame auf dem Bildschirm, und zwar unterhalb der übergebenen Komponente *com* nach links ausdehnend. Zusätzlich versucht der FlitListFrame, mindestens so weit zu breit wie das angegebene TextColField *tcf* zu sein. Diese Position wird dabei nur als Empfehlung betrachtet, von der abgewichen wird, falls der FlitListFrame sonst nicht mehr vollständig auf dem Bildschirm sichtbar wäre.

+requestFocus(): void

Veranlasst den FlitListFrame, den Eingabe-Focus in die Liste zu holen.

+windowActivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces; holt den Focus in die Liste.

+windowClosed (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowClosing (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowDeactivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowDeiconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowIconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowOpened (e: WindowEvent): void

Implementation des WindowListener-Interfaces; holt den Focus in die Liste.

7.33 Klasse FlitLoginFrame

extends java.awt.Frame

implements ActionListener, ItemListener,KeyListener

Der FlitLoginFrame (Verzeichnis `ui/`) ist das Login-Fenster der Maske. Darin soll der Benutzer Namen und Passwort angeben, und mit diesen Daten wird ein Login in der Datenbank versucht. Gelingt dieser, wird die Programmkontrolle an einen AblaufVerwalter weitergegeben.

Konstruktoren:

+FlitLoginFrame (inAnApplet: boolean)

Erzeugt einen FlitLoginFrame und zeigt ihn an. Der Wert des Parameters *inAnApplet*, der anzeigt, ob das Programm als Applet läuft, wird an die Klassen, deren Verhalten sich nach dieser Information richtet, weitergegeben.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListeners.

+itemStateChanged (e: ItemEvent): void

Implementation des ItemListeners.

+keyPressed (e: KeyEvent): void

Implementation des KeyListeners.

+keyReleased (e: KeyEvent): void

Implementation des KeyListeners; leer.

+keyTyped (e: KeyEvent): void

Implementation des KeyListeners; leer.

-trylogin(): void

Versucht den Login mit den momentan eingestellten bzw. gelesenen Informationen und gibt ggf. eine Fehlermeldung in der Statuszeile des FlitLoginFrames aus.

-loginException (e: Exception): void

Überprüft im Falle einer Exception während des Logins, ob diese auf eine ungültige Username/Password-Kombination zurückzuführen ist, und gibt abhängig davon eine Fehlermeludung in der Statuszeile des FlitLoginFrames aus.

7.34 Klasse FlitSpeedBar

extends java.awt.Panel

Die FlitSpeedBar (Verzeichnis `shared/ui/`) ist ein Panel, das die „speedbar“ Button-Leiste enthält. Es zeigt entweder die Button-Leiste „speedbar“, einen Textstring „stop“ oder eine leere Fläche. Die Beschriftungen, die intern benutzten *ActionCommands* und die Reihenfolge der Buttons wird aus dem ResourceBundle Sprache.properties geladen.

Konstruktoren:

+FlitSpeedBar (buttons: Hashtable, rb: ResourceBundle)

Erzeugt eine FlitSpeedBar; die Buttons, die aus dem ResourceBundle *rb* gelesen werden, werden im Hashtable *buttons* für späteren Zugriff gespeichert.

Methoden:

+displayBar (barname: String): void

Zeigt die gewünschte Button-Leiste mit dem Namen *barname* an. Mögliche Namen sind „speedbar“, „stop“ und „empty“.

+setVisibleAbbruch (visible: boolean): void

Setzt den Button „Abbruch“ auf sichtbar oder unsichtbar.

7.35 Klasse FlitStrich

extends java.awt.Canvas

Der FlitStrich (Verzeichnis `shared/ui/`) ist ein senkrechter Strich. Er kann durchsichtig, schwarz oder rot sein, oder auch ein 3D-Aussehen annehmen. Die Breite des Striches ist fixiert, in der Höhe kann er sich beliebig ausdehnen. Er wird in den meisten Fällen dazu benutzt, den Status eines Textfeldes, zu dem er gehört, anzuzeigen.

Attribute:

+$ERROR: int

Konstante, die `setMode` übergeben werden kann, um den FlitStrich einen Fehler signalisieren zu lassen.

+$LOOK_3D: int

Konstante, die `setMode` übergeben werden kann, um den FlitStrich „abzusenken“; dies wird für Trennstriche benutzt, und wenn ein Feld als deaktiviert gekennzeichnet werden soll.

+$MODIFIED: int

Konstante, die `setMode` übergeben werden kann, um den FlitStrich ein modifiziertes Feld markieren zu lassen.

+$OK: int

Konstante, die `setMode` übergeben werden kann, um den FlitStrich unsichtbar zu machen.

-col: Color

Die momentane Farbe des FlitStriches.

Konstruktoren:

+FlitStrich()

Erzeugt einen (anfangs unsichtbaren) FlitStrich.

+FlitStrich (i: int)

Erzeugt einen FlitStrich in der gewüschten Farbe.

Methoden:

+getMaximumSize(): Dimension

Liefert die maximale Dimension dieses FlitStriches.

+getMinimumSize(): Dimension

Liefert die minimale Dimension dieses FlitStriches.

+getPreferredSize(): Dimension

Liefert die gewünschte Dimension dieses FlitStriches.

+paint (g: Graphics): void

Zeichnet den FlitStrich.

+setMode (i: int): void

Setzt die Farbe des FlitStriches; die möglichen Modi können mittels der öffentlichen Konstanten dieser Klasse erreicht werden.

7.36 Klasse FlitTextArea

extends java.awt.TextArea

Die FlitTextArea (Verzeichnis `shared/ui/`) ist eine TextArea, die den Focus nur dann akzeptiert, wenn sie editierbar ist.

Konstruktoren:

+FlitTextArea (rows: int, columns: int)

Erzeugt eine TextArea mit *rows* Zeilen und *columns* Spalten.

Methoden:

+isFocusTraversable(): boolean

Gibt zurück, ob die TextArea editierbar ist, denn genau dann soll sie den Fokus akzeptieren.

7.37 Klasse FlitTextField

extends java.awt.TextField

Das FlitTextField (Verzeichnis `shared/ui/`) ist ein TextField, das den Input-Focus nur dann akzeptiert, wenn es editierbar ist.

Konstruktoren:

+FlitTextField (visible: int)

Erzeugt ein TextField mit *visible* Zeichen.

Methoden:

+isFocusTraversable(): boolean

Gibt zurück, ob das TextField editierbar ist, denn genau dann soll sie den Fokus akzeptieren.

7.38 Klasse FlitTextPanel

extends java.awt.Panel

Das FlitTextPanel (Verzeichnis `shared/ui/`) ist das Panel, das sämtliche Ein- und Ausgabefelder für Texte enthält. Die Anordnung, Typen und Beschriftungen der Felder wird aus einem ResourceBundle geladen; üblicherweise dem ResourceBundle `Sprache.properties`.

Konstruktoren:

+FlitTextPanel (buttons: Hashtable, textcolitems: Hashtable, rb: ResourceBundle, kl: KeyListener)

Erzeugt ein neues FLITTEXTPANEL. Dabei sind *buttons* und *textcolitems* Hashtables, die die erzeugten Buttons bzw. TEXTCOLITEMS (mit ihren *ActionCommands* als Schlüssel) speichert. Die Anordnung, Typen und Beschriftungen der Eingabefelder werden aus dem ResourceBundle *rb* gelesen. Der KeyListener *kl* wird an alle Komponenten außer den Buttons, die in *buttons* eingetragen werden, angehängt.

-createTextColXXX (cont: Container, gb: GridBagLayout, name: String, uiname: String, [...]): TextColXXX

Für jeden Typ von TEXTCOLITEM gibt es eine solche Methode, die eine Instanz der entsprechenden Klasse erzeugt und mittels der entsprechenden `addToGB`-Methode dem Panel hinzufügt. Das neue TEXTCOLITEM wird dann zurückgegeben.

-addToGB (cont: Container, gb: GridBagLayout, tc$X$: TextColXXX, last: boolean): void

Für jeden Typ von TEXTCOLITEM gibt es eine solche Methode, die das übergebene TEXTCOLITEM *tcX* dem Container *cont* hinzufügt. Der Wert *last* gibt an, ob es die aktuelle Zeile bis zum Ende füllen soll.

7.39 Klasse KRITERIEN

extends java.lang.Object

Die Klasse KRITERIEN (Verzeichnis `dok/`) enthält alle Felder, für die Suchkriterien eingegeben werden können. Jeder neue Wert für ein Feld wird sofort mit Hilfe des entsprechenden EXPRPARSERS geparst. Falls ein Fehler auftritt, kann man die (vermeintliche) Fehlerposition und einen Fehlertext holen. Man kann das automatische Einkleiden der Kriterien in WildCards aktivieren und deaktivieren, und man kann einstellen, ob die Suchkriterien Groß/Kleinschreibung unterscheiden sollen oder nicht.

Nicht aufgeführt werden hier die „public static final“ Konstanten, die die Namen der Spalten in der Datenbankrelation bezeichnen und erst durch das Instantiieren in diese Klasse eingetragen werden.

Attribute:

+$fieldnames: String[]

Ein Array mit allen Spaltennamen der Datenbankrelation.

Konstruktoren:

+Kriterien()

Erzeugt eine neue Instanz ohne eingetragene Werte; für jedes Feld wird ein Parser erzeugt.

Methoden:

+clear(): void

Löscht alle enthaltenen Kriterien.

+get (feld: String): String

Holt den Wert des Feldes mit dem Namen *feld*; liefert `Null` wenn kein Wert gespeichert ist.

+getErrorPos (feld: String): int

Holt die (vermeintliche) Position des Parse-Fehlers im Feld mit dem Namen *feld*. Die Position ist >= 0 und <= length für einen Wert der Länge „length“. Man beachte, dass „= length“ keine gültige Position innerhalb eines Strings darstellt; in diesem Fall war der Ausdruck vermutlich vorzeitig zu Ende. War gar kein Fehler aufgetreten, ist die Rückgabe undefiniert.

+getErrorString (feld: String): String

Holt die Klartext-Fehlermeldung des Parse-Fehlers des Feldes mit dem Namen *feld*, oder `Null` wenn kein Fehler aufgetreten war oder kein Wert gesetzt ist.

+getKriterien(): Enumeration

Liefert eine Aufzählung aller Feldnamen, zu denen ein Wert gespeichert ist.

+getSQL (feld: String): String

Holt die SQL-Übersetzung des Feldes mit dem Namen *feld*. Die Spaltennamen der Felder aus der Datenbanktabelle haben das Präfix „d.“. Liefert den leeren String, wenn kein Wert in diesem Feld gespeichert ist, oder `Null` wenn beim Parsen des Wertes ein Fehler aufgetreten ist.

+hasSQLParser (feld: String): boolean

Tested, ob zu dem Feld mit dem Namen *feld* ein SQLParser existiert. Falls nicht, darf für dieses Feld keine der Methoden `getSQL`, `getErrorString` oder `getErrorPos` aufgerufen werden.

+isEmpty(): boolean

Gibt zurück, ob sämtliche Werte dieses Kriteriensatzes leer sind.

+set (feld: String, wert: String): boolean

Setzt den Wert des Feldes *fesl* auf *wert*; dabei löschen `Null` und der leere String den Feldinhalt. Der neue Wert wird sofort geparst; falls dabei ein Fehler auftritt, wird `False` zurückgegeben, ansonsten `True`.

+setAutoWildCards (auto: boolean): void

Aktiviert oder deaktiviert das automatische Einkleiden der Kriterien in WildCards je nach Wert von *auto*. Default ist `False`, d. h. deaktiviert. Dies bezieht sich auf alle Felder gleichzeitig, wird allerdings nur bei solchen wirksam, bei denen es im zugehörigen ExprParser zugelassen ist.

+setCaseSensitive (casesensitive: boolean): void

Aktiviert oder deaktiviert die Beachtung der Groß/Kleinschreibung je nach Wert von *casesensitive*. Default ist `False`, d. h. Groß/Kleinschreibung wird nicht unterschieden.

-add*XXX* (name: String): void

Für die einzelnen Wert-Typen gibt es private Methoden, die einen neuen Expr*XXX*Parser erzeugen, mit der Spalte *name* verbinden und in der Klasse speichern.

7.40 Klasse ListAssistantFrame

extends java.awt.Frame

implements WindowListener, ActionListener, ItemListener, TextColListener, ListFrameListener

Der ListAssistantFrame (Verzeichnis `shared/ui/`) ist ein selbständiges Fenster der Masken, das einen Assistenten zur Erstellung einer Liste aus einem Suchergebnis bereitstellt.

Konstruktoren:

+ListAssistantFrame (lal: ListAssistantListener, font: Font, all: boolean, editable: boolean, wrap: boolean, separate: boolean, isApplet: boolean)

Erzeugt einen neuen ListAssistantFrame, der den Font *font* benutzt und die entsprechenden Events an den ListAssistantListener *lal* weitergibt. Die anderen Parameter geben an, ob zu Beginn „gesamtes Suchergebnis“ (*all*), „editierbar“ (*editable*), „automatischer Zeilenumbruch“ (*wrap*) oder „Leerzeile einfügen“ (*separate*) selektiert sein sollen. Der letzte Parameter gibt an, ob das Programm als Applet läuft.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+displayAndToFront(): void

Macht diesen ListAssistantFrame sichtbar und bringt ihn nach vorne.

+enableOK (ok: boolean): void

Aktiviert oder deaktiviert den OK-Knopf je nach Wert von *ok*.

+enableSave (saveable: boolean): void

Aktiviert oder deaktiviert den Save-Knopf je nach Wert von *saveable*.

+executeListCommand (ac: String, flf: FlitListFrame, name: String, value: String): void

Implementation des ListFrameListener-Interfaces.

+getFormat(): String

Holt den Inhalt des Format-Eingabefeldes.

+itemStateChanged (e: ItemEvent): void

Implementation des ItemListener-Interfaces.

+pleaseCenterOver (parent: Frame): void

Positioniert den ListAssistantFrame so auf dem Bildschirm, dass er zentriert über dem Frame *parent* erscheint, ohne sich jedoch über den Bildschirm hinaus zu erstrecken.

+pleaseDispose(): void

Veranlasst den ListAssistantFrame, sich zu schließen und alle Resourcen freizugeben. Darf mehrfach aufgerufen werden – alle Aufrufe nach dem ersten sind dann wirkungslos.

+pleasePositionHere (p: Point): void

Positioniert den ListAssistantFrame auf dem Bildschirm. Die gegebene Position wird dabei nur als Empfehlung betrachtet, von der abgewichen wird, falls der ListAssistantFrame sonst nicht mehr vollständig auf dem Bildschirm sichtbar wäre.

+requestFocus(): void

Veranlasst den ListAssistantFrame, den Eingabe-Fokus in sein Fenster zu holen.

+setFirstLast (first: String, last: String): void

Setzt die Texte *first* und *last* in die Felder „von“ bzw. „bis“ für die Auswahl des Bereiches innerhalb des SuchErgebnisses ein.

+setFormat (format: String): void

Setzt den Text *format* in das Format-Eingabefeld ein.

+setNumberOfDocuments (n: int): void

Setzt die Angabe, wieviele Dokumente sich im SuchErgebnis befinden, auf n. Negative Zahlen haben dieselbe Bedeutung wie bei der Methode `size()` im SuchErgebnis.

+textColValueChanged (source: TextColItem, name: String, text: String): void

Implementation des TextColListener-Interfaces, um den ListAssistantListener benachrichtigen zu können.

+windowActivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowClosed (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowClosing (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowDeactivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowDeiconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowIconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowOpened (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

7.41 Interface ListAssistantListener

Der ListAssistantListener (Verzeichnis `shared/ui/`) ist ein Interface zur Kommunikation zwischen einem ListAssistantFrame und einem Arbeitsablauf, der insbesondere auf das Beenden des ListAssistantFrames reagieren muss.

Methoden:

+listAssistant_executeCommand (ac: String): void

Wird aufgerufen, wenn der Befehl *ac* in einem ListAssistantFrame ausgeführt werden soll.

+listAssistant_textColValueChanged (source: TextColItem, name: String, text: String): void

Wird aufgerufen, wenn sich der Wert im TextColItem *source* mit dem Namen *name* geändert hat; der neue Inhalt ist nun *text*.

7.42 Klasse ListendarstellungFrame

extends java.awt.Frame

implements WindowListener, ActionListener

Der ListendarstellungFrame (Verzeichnis `shared/ui/`) ist ein Fenster der Masken, das eine mit dem Listenassistenten erstellten Ergebnisliste anzeigt.

Konstruktoren:

+ListendarstellungFrame (s: String, font: Font, editable: boolean, wrap: boolean, inAnApplet: boolean)

Erzeugt einen neuen ListendarstellungFrame, der den Font *font* benutzt und den Text *s* anzeigt. Die TextArea ist editierbar wenn *editable* auf `True` gesetzt ist, und führt automatische Zeilenumbrüche aus, wenn *wrap* auf `True` gesetzt ist. Der letzte Parameter *inAnApplet* gibt an, ob das Programm als Applet läuft.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+setDefaultFileName (defaultFileName: String): void

Setzt den Namen, unter dem die Liste gespeichert wird, solange vom Benutzer kein anderer Wert eingegeben wird, auf *defaultFileName*.

+windowActivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowClosed (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowClosing (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den Frame.

+windowDeactivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowDeiconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowIconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowOpened (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

7.43 Interface ListFrameListener

Der ListFrameListener (Verzeichnis `shared/ui/`) ist ein Interface zur Kommunikation zwischen einem FlitListFrame und einem Arbeitsablauf.

Methoden:

+executeListCommand (ac: String, flf: FlitListFrame, name: String, value: String): void

Wird aufgerufen, wenn ein Befehl in einem FlitListFrame ausgeführt werden soll; dabei ist *ac* das ActionCommand, *flf* der FlitListFrame, von dem der Befehl kommt, *name* der Name des ListFrames und *value* der Wert, den der Benutzer ausgesucht hat.

7.44 Klasse ListManager

extends java.lang.Object

Der ListManager (Verzeichnis `dok/`) kann aus einem SuchErgebnis eine Übersicht in Form eines Strings erstellen. Der String enthält in jeder Zeile einen Datensatz des Suchergebnisses. Start- und Endposition innerhalb des Suchergebnisses sind frei wählbar. Ebenso ist frei wählbar, welche Spalten die Liste enthalten soll, und deren Reihenfolge.

Die abstrakte interne Klasse Part wird zum schnellen Erzeugen des Listentextes benutzt. Der Listentext besteht aus vielen Zeilen, von denen jede aus mehreren Teilen (Parts) besteht. Ein solcher Teil kann ein Feld (Field) eines Datensatzes oder ein Trennzeichen (Delimiter) sein.

Vor dem eigentlichen Erzeugen des Listentextes wird der Formatstring, der dem Konstruktor übergeben wird, geparst und in eine Folge von Parts übersetzt. Diese Folge enthält dann üblicherweise abwechselnd Fields und Delimiter. Für diese Übersetzung werden die Übersetzungstabellen „tt“ und „tp“ benutzt. Dabei enthält „tt[x]“ alle Möglichkeiten für eine vom Benutzer eingegebene Zeichenfolge, die ein Feld darstellt und mit dem Buchstaben x+'a' beginnt (es findet also ein Hashing über den ersten Buchstaben statt). Hat der Parser in „tt[x][y]“ einen String gefunden, der in der Eingabe vorkommt, so findet er in „tp[x][y]“ den zugeordneten 'Part'. Alle Characters 'dazwischen' werden in Delimitern zusammengefasst.

Attribute:

#pos_ende: int

Position des letzten Datensatzes, der in die Liste aufgenommen werden soll; der Wert -1 bedeutet, dass alle Datensätze bis zum Ende aufgenommen werden sollen.

#pos_start: int

Position des ersten Datensatzes; hierbei bezeichnet 0 den ersten Datensatz des SuchErgebnisses.

#prefix: String

Der Text, der der eigentlichen Liste vorangestellt werden soll.

#se: SuchErgebnis

Das SuchErgebnis, aus dem die Liste zusammengestellt werden soll.

#sp: Part[]

Dieser Array gibt an, aus welchen Teilen der Ausgabestring besteht.

#sp_max: int

Gibt an, aus wievielen Spalten der Ausgabestring besteht.

#suffix: String

Der Text, der nach der eigentlichen Liste ausgegeben werden soll.

Konstruktoren:

+ListManager (se: SuchErgebnis, spalten: String)

Erzeugt einen ListManager zum SuchErgebnis *se*. Das Format des Ausgabestrings wird mit Spaltenbezeichnern und Trennzeichen in *spalten* angegeben.

Methoden:

#appendWithNewLine (b: StringBuffer, s: String): void

Hängt den String *s* an den StringBuffer *b* an, gefolgt von einem „*newline*“, falls *s* nicht bereits mit einem „*newline*“ aufhört.

+getList (separateByLine: boolean): String

Erzeugt eine Liste aus dem SuchErgebnis für den gewählten Bereich; dies ist einen aufwendige Operation. Die Liste steht in einem String, der zurückgegeben wird. Ist *separateByLine* auf `True` gesetzt, so werden zwei Datensätze durch eine Freizeile getrennt.

+setPrefix (prefix: String): void

Setzt den Text, der der eigentlichen Liste vorangestellt werden soll; darf `Null` sein.

+setRange (first: int, last: int): void

Setzt den Bereich, der in die Liste aufgenommen werden soll; als Default wird die gesamte Liste aufgenommen. *first* gibt die Position des ersten Datensatzes im SuchErgebnis an, der in die Liste aufgenommen werden soll, dabei ist 0 der erste Datensatz des SuchErgebnisses; *last* gibt die Position des letzten Datensatzes im SuchErgebnis an, der in die Liste aufgenommen werden soll, dabei bedeutet der Wert -1, dass die Datensätze bis zum Ende des SuchErgebnisses Beachtung finden sollen.

+setSuffix (suffix: String): void

Setzt den Text, der der eigentlichen Liste folgt; darf `Null` sein.

-parseString (s_orig: String): Part[]

Parst den Formatstring *s_orig* und erzeugt einen entsprechenden Array von Parts, der zurückgegeben wird.

7.45 Klasse LoginManager

extends java.lang.Object

Der LoginManager (Verzeichnis `net/`) ist für den Login-Vorgang zuständig. Er reicht Usernamen und Passwort an die Datenbank weiter und stellt fest, ob Schreibrechte gewährt werden. Geligt der Login, wird danach ein Maskenfenster geöffnet.

Konstruktoren:

+LoginManager()

Default-Konstruktor; leer, da die Klasse nur statisch benutzt wird.

Methoden:

+$login (driver: String, url: String, name: String, pass: String, isGast: boolean, inAnApplet: boolean): void

Führt den Login-Vorgang durch. Der Aufrufer sollte vorher im ABLAUFVERWALTER den gewünschten Font für das Maskenfenster gesetzt haben. Die Verbindung an die Datenbank wird mittels des Drivers *driver* und der URL *url* aufgebaut, und die Authentifizierung wird mittels *name* und *pass* vorgenommen. Ist *isGast* auf `False` gesetzt, wird ausserdem auf Schreibrechte geprüft. Der Parameter *inAnApplet* gibt an, ob das Programm als Applet läuft.

7.46 Klasse RBMANAGER

extends java.lang.Object

Der RBMANAGER (Verzeichnis `shared/net/`) lädt und verwaltet die von einer Maske benötigten ResourceBundles. Darüber hinaus werden häufig benötigte Funktionen auf ResourceBundles zur Verfügung gestellt.

Attribute:

-$rbCache: Hashtable

Cached die geladenen ResourceBundles unter ihrem Namen.

-$inAnApplet: boolean

Gibt an, ob das Programm als Applet läuft.

-$hostname: String

Gibt den Hostnamen der Datenbank, falls das Programm als Applet läuft.

-$alterLoc: Vector

Ein Vektor mit alternativen URLs, unter denen nach ResourceBundles gesucht werden kann.

Konstruktoren:

+RBManager()

Default-Konstruktor; leer, da die Klasse nur statisch benutzt wird.

Methoden:

+$appendToLocalRB (rbname: String, p: String, value: String): boolean

Fügt an einen Vektor in einem lokal vorhandenen ResourceBundle mit dem Namen *rbname*, der durch das Präfix *p* identifiziert wird, den neuen Wert *value* an.

+$appendToVectorFromRB (rb: ResourceBundle, p: String, s: String, v: Vector): void

Liest einen Vektor aus dem ResourceBundle *rb*, der durch das Präfix *p* identifiziert und das Suffix *s* ausgewählt wird, d. h. es werden die Werte `p+„1\+s`, `p+„2\+s`, usw. gelesen und an den Vektor *v* angehängt.

+$appendToVectorFromRB (rb: ResourceBundle, p: String, v: Vector): void

Liest einen Vektor aus dem ResourceBundle *rb*, der durch das Präfix *p* identifiziert wird, d. h. es werden die Werte `p+„1\`, `p+„2\`, usw. gelesen und an den Vektor *v* angehängt.

+$getBooleanFromRB (rb: ResourceBundle, s: String): boolean

Liest einen Booleanwert aus aus dem ResourceBundle *rb*, der durch den Kennzeichner *s* identifiziert wird. Der Wert muss eine der beiden Zeichenketten „true“ oder „false“ sein.

+$getHashtableFromRB (rb: ResourceBundle, s: String, key: String, value: String): Hashtable

Liest einen Hastable aus dem ResourceBundle *rb*, der durch den Kennzeichner *s* identifiziert wird. Das Key-Suffix *key* legt fest, welche Zeilen im ResourceBundle als Schlüsselwerte erkannt werden, und das Wert-Suffix *value* erfüllt diese Funktion für die Einträge.

+$getIntFromRB (rb: ResourceBundle, s: String): int

Liest einen Integerwert aus dem ResourceBundle *rb*, der durch den Kennzeichner *s* identifiziert wird. Falls der Zahlenwert im ResourceBundle keine gültige Dezimalzahl darstellt und das Parsen eine NumberFormatException wirft, wird diese abgefangen und ignoriert. Es wird dann einfach der Wert 1 zurückgegeben.

+$getRB (rbname: String): ResourceBundle

Liefert ein ResourceBundle mit dem Namen *rbname*. Jedes einmal angeforderte ResourceBundle wird über seinen Namen in einem Hashtable gechachet, damit spätere Zugriffe schneller erfolgen.

+$getRB (rbname: String, mustFind: boolean): ResourceBundle

Liefert ein ResourceBundle mit dem Namen *rbname*, allerdings wird nur dann eine Exception geworfen, falls es nicht gefunden werden konnte, wenn *mustFind* auf `True` gesetzt ist.

+$getVectorFromRB (rb: ResourceBundle, p: String): Vector

Liest einen Vektor aus dem ResourceBundle *rb*, der durch das Präfix *p* identifiziert wird, d. h. es werden die Werte `p+„1\`, `p+„2\`, usw. gelesen und in einem Vektor gespeichert.

+$getVectorFromRB (rb: ResourceBundle, p: String, s: String): Vector

Liest einen Vektor aus dem ResourceBundle *rb*, der durch das Präfix *p* identifiziert und das Suffix *s* ausgewählt wird, d. h. es werden die Werte `p+'1'+s`, `p+'2'+s`, usw. gelesen und in einem Vektor gespeichert.

+$setAlterLoc (loc: Vector): void

Setzt einen Vektor von alternativen URLs, um nach ResourceBundles zu suchen. Der RBManager verwendet dann u. U. weitere Methoden, um ResourceBundles zu laden.

+$setInAnApplet (hostname: String): void

Teilt dem RBManager mit, dass das Programm als Applet läuft, und setzt den Hostnamen auf *hostname*. Der RBManager verwendet dann u. U. andere Methoden, um ResourceBundles zu laden.

-$createRB (name: String): ResourceBundle

Holt ein ResourceBundle, wobei das zuerst lokal und dann an den einzelnen URLs des Vektors „alterLoc“ versucht wird.

7.47 Klasse SaveFrame

extends java.awt.Frame

implements ActionListener

Der SaveFrame (Verzeichnis `shared/ui/`) ist ein Fenster der Masken, das aufgerufen wird, wenn ein Text gespeichert werden soll.

Konstruktoren:

+SaveFrame (parent: Frame, defaultFileName: String, textToSave: String, errorText: String)

Erzeugt einen neuen SaveFrame, der sich über dem Frame *parent* befindet und den *defaultFileName* anzeigt. Wenn der Benutzer den Filenamen akzeptiert, wird der Text *textToSave* gespeichert; tritt dabei ein Fehler auf, wird der Text *errorText* ausgegeben.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

7.48 Klasse SQLAssistant

extends java.lang.Object

Der SQLAssistant (Verzeichnis `dok/`) wird von einem Arbeitsablauf benutzt und erledigt die SQL-bezogene schwierige Arbeit.

Attribute:

-$s_readDokument: String[]

Die Namen der Spalten in der Datenbankrelation für das Auslesen der Datensätze. Die Reihenfolge ist hiermit festgelegt.

-$s_readColums: String

Ein String mit allen Spaltennamen der Tabelle, durch Kommata getrennt und mit dem Präfix „d.“ versehen.

-$sa_insertNewDokument: String[]

Die Namen der Spalten der Datenbankrelation in der korrekten Reihenfolge für das Einfügen eines neuen Datensatzes. Beginnt der String mit '+', so wird nach vom Programm generierten Werten geschaut, beginnt er mit '-', so wird der Wert `Null` eingetragen.

-$sa_updateDokument: String[]

Die Namen der Spalten der Datenbankrelation für das Aktualisieren eines modifizierten Datensatzes.

Konstruktoren:

+SQLAssistant (con: Connection)

Erzeugt einen neuen SQLAssistant zu der gegebenen Connection *con*.

Methoden:

+checkForWritePermissions(): String

Liest den nächsten Wert aus der entsprechenden Sequence in der Datenbank. Gibt den Wert als String zurück, falls das Lesen erfolgreich war; andernfalls wird ein String zurückgegeben, der als erstes Zeichen ein '*' enthält, und zwar „*permission denied“, wenn das Lesen eine Exception geworfen hat (der Account also keine Schreibrechte hat), oder „*access error“, falls ein anderer Zugriffsfehler aufgetreten ist.

+commit(): void

Führt ein Datenbank-Commit durch; wirft eine SQLException, wenn dabei ein Datenbank-Fehler auftritt.

#commitDokument (dok: DokumentDatensatz): void

Commitet den DokumentDatensatz *dok*. Ist das Dokument als gelöscht markiert, wird es vollständig aus der Datenbank gelöscht. Ist es modifiziert, werden die neuen Werte in die Datenbank eingetragen. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

+createDokumentDatensatz (rs: ResultSet): DokumentDatensatz

Erzeugt einen DokumentDatensatz, indem die Daten aus dem ResultSet *rs* gelesen werden. Der Cursor im ResultSet muss in der gewünschten auszulesenden Zeile stehen. Das ResultSet muss aus einer Anfrage stammen, bei der die Spalten genau in der Reihenfolge stehen, die durch `getDokumentColumnNames` vorgegeben ist. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

+createSuchErgebnis (sqlcmd: String): SuchErgebnis

Führt die Datenbankanfrage *sqlcmd* durch und bereitet die Ergebnismenge in einem SuchErgebnis auf, das dann zurückgegeben wird. Die SQL-Anfrage muss vom Aufrufer zusammengestellt worden sein, wobei die Reihenfolge der Spaltennamen entscheidend ist; sie sollten von der Methode `getDokumentCokumnNames` erzeugt worden sein. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

+createUniqueList (colname: String): Vector

Erzeugt zur Spalte *colname* eine sortierte Liste aller in der Datenbank vorkommenden Werte mittels einer SQL-Anfrage. Wirft eine SQLException, falls dabei ein Datenbank-Fehler auftritt.

+getDokumentColumnNames(): String

Liefert einen String mit allen Spaltennamen der Datenbankrelation, durch Kommata getrennt. Vor jedem Spaltennamen ist das Präfix „d.“ gesetzt. Die Reihenfolge der Spaltennamen ist bei allen lesenden SQL-Anfragen bindend, da sich andere Methoden auf die Reihenfolge der Spalten im ResultSet verlassen.

+getTableName(): String

Liefert einen String mit dem Namen der Haupttabelle.

+getTableOwnerName(): String

Liefert einen String mit dem Namen des Benutzers, dem die Tabellen in der Datenbank gehören.

+insertNewDokument (dok: DokumentDatensatz): String

Fügt ein neues Dokument in die Datenbank ein. Ermittelt zum Einfügen die Werte, die das Programm selbst generiert. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

+lockDokument (dd: DokumentDatensatz): int

Sperrt einen DokumentDatensatz in der Datenbank für exklusiven Zugriff (select for update nowait) und prüft, ob die aktuellen Daten in der Datenbank noch mit den Originaldaten des Datensatzes übereinstimmen. In Abhängigkeit vom Ausgang des Sperrversuchs und des Datenvergleichs wird der DokumentDatensatz in einen bestimmten Zustand gesetzt und ein entsprechender Integer-Wert zurückgegeben:

Faktenlage	DokumentDatensatz	Wert
Sperren erfolgreich, Daten unverändert	exclusiv	1
Sperren erfolgreich, Daten wurden zwischenzeitlich geändert oder gelöscht	exclusiv, unmodifizierbar	0
Sperren fehlgeschlagen	unmodifizierbar	-1

Der Rückgabewert ist also genau dann $<= 0$, wenn der DokumentDatensatz jetzt nicht mehr modifizierbar ist, und genau dann $>= 0$, wenn der DokumentDatensatz jetzt exklusiv ist.

+$quote (s: String): String

Ersetzt jeden vorkommenden Apostroph durch zwei Apostrophe.

+restore (se: SuchErgebnis): void

Setzt das SuchErgebnis *se* komplett in den Originalzustand zurück und verwirft alle daran vorgenommenen Modifikationen.

+rollback(): void

Führt ein Datenbank-Rollback durch; wirft eine SQLException, wenn dabei ein Datenbank-Fehler auftritt.

+$setGast (gast: boolean): void

Aktiviert oder deaktiviert den Gast-Modus ja nach Wert von *gast*.

-deleteDokument (dok: DokumentDatensatz): void

Löscht das Dokument *dok* aus der Datenbank. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

-updateDokument (dok: DokumentDatensatz): void

Aktualisiert den modifizierten DokumentDatensatz *dok* in der Datenbank. Wirft eine SQLException, falls ein Datenbank-Fehler auftritt.

7.49 Klasse SQLQuery

extends java.lang.Object

Die Klasse SQLQuery (Verzeichnis `dok/`) wird von einem Arbeitsablauf benutzt. Sie kann aus gegebenen Kriterien erzeugt, durch weitere Kriterien verfeinert oder erweitert werden, und gibt die entstehende Anfrage SQL-konform zurück.

Konstruktoren:

+SQLQuery (Kriterien k)

Erzeugt einen neue SQLQuery aus den Kriterien *k*. Falls das Kriterien-Objekt Werte enthält, die nicht korrekt geparst werden konnten, werden diese Werte ignoriert.

Methoden:

+add (Kriterien k): void

Erweitert die SQLQuery durch weitere Kriterien. Falls das Kriterien-Objekt Werte enthält, die nicht korrekt geparst werden konnten, werden diese Werte ignoriert.

+getSQLCmd (orderby: String): String

Liefert einen String mit dem SQL-Befehl. Der Parameter *orderby* enthält einen oder mehrere, durch Kommata getrennte Spaltennamen, nach denen sortiert werden soll.

+refine (Kriterien k): void

Verfeinert die SQLQuery durch weitere Kriterien. Falls das Kriterien-Objekt Werte enthält, die nicht korrekt geparst werden konnten, werden diese Werte ignoriert.

+setWhere (kbed: String): void

Bietet die Möglichkeit, eine spezielle WHERE-Bedingung von Hand einzugeben. Vorher eingegebene Kriterien werden überschrieben.

7.50 Klasse SuchErgebnis

extends java.lang.Object

Das SuchErgebnis (Verzeichnis `dok/`) stellt eine komfortable Schnittstelle zur Ergebnismenge einer Datenbankanfrage dar. Es bietet einen SuchErgebnis-Cursor zum Navigieren im SuchErgebnis; gespeichert und zurückgegeben werden DokumentDatensätze. Das SuchErgebins kann mittels `commit` in die Datenbank eingetragen werden, dabei werden alle Modifikationen und Löschungen in die Datenbank übernommen. Solange das SuchErgebnis dadurch nicht leer wird, d. h. alle Datensätze gelöscht wurden, bleibt es weiterhin geöffnet und kann benutzt werden.

Das SuchErgebnis liest nur die Datensätze aus dem zugrundeliegenden ResultSet, die tatsächlich vom Aufrufer angefordert werden. Falls das ResultSet noch nicht bis zum Ende gelesen wurde und gerade gar keine Datensätze im SuchErgebnis gecachted sind, wird automatisch ein Datensatz gelesen und gecacht.

Das SuchErgebnis ist thread-safe. Man kann also in einem Hintergrund-Thread mit Hilfe eines SuchErgebnisCursors nebenbei Datensätze anfordern, die dann dadurch bereits im SuchErgebnis gecacht zur Verfügung stehen, wenn der Benutzer im Ergebnis blättert. Man sollte das Im-Voraus-Lesen allerdings nicht übertreiben, weil das Bandbreite der Datenbankanbindung verbraucht.

Achtung: Das SuchErgebnis ist thread-safe; ob die Datenbanktreiber thread-safe sind, ist eine andere Frage!

Attribute:

-rs: ResultSet

Das zugehörtige ResultSet.

-rsmd: ResulSetMetaData

Die Meta-Daten des ResultSets.

-closed: boolean

Gibt an, ob dieses SuchErgebnis geschlossen ist.

-erg: Vector

Vektor zum Speichern der DokumentDatensätze.

-endReached: boolean

Gibt an, ob bereits das Ende im ResultSet erreicht wurde.

Konstruktoren:

+SuchErgebnis (rs: ResultSet, stmt: Statement, sqla: SQLAssistant)

Erzeugt ein neues SuchErgebnis zum ResultSet *rs*. Das übergebene Statement *stmt* wird geschlossen, nachdem das ResultSet geschlossen wird, ansonsten wird es nicht benutzt, und darf auch `Null` sein. Das SuchErgebnis versucht sofort, den ersten DokumentDatensatz vom ResultSet einzulesen. Falls das gegebene ResultSet leer ist, schließt sich das SuchErgebnis sofort selbst.

Methoden:

+close(): void

Schließt das SuchErgebnis und seine Datenbank-Resourcen wie das ResultSet und das Statement. Beim Schließen eventuell auftretende SQLExceptions werden abgefangen und ignoriert.

+commit(): void

Commited das SuchErgebnis, indem jedes Dokument einzeln aus der Datenbank gelöscht wird, bzw. Modifikationen übertragen werden. Danach wird ein Datenbank-Commit ausgeführt. Konnte dies erfolgreich ausgeführt werden, sind damit alle Modifikationen endgültig in die Datenbank übernommen, und alle DokumentDatensätze sind wieder im nicht-exklusiven Zustand.
Behandlung von Datenbank-SQLExceptions: Falls beim abschließenden commit eine SQLException auftritt, wird das SuchErgebnis überhaupt nicht verändert. Andernfalls wird jedes als gelöscht markierte Dokument, bei dessen Löschen in der Datenbank keine SQLException aufgetreten ist, aus dem SuchErgebnis entfernt. Bei jedem Dokument, das keine SQLException beim Aktualisieren in der Datenbank hervorgerufen hat, wird die Methode `commit` des DokumentDatensatzes aufgerufen.
Referenzen auf DokumentDatensätze müssen nach einem `commit` verworfen und neu aus dem SuchErgebnis geholt werden, da sie sich geändert haben können.
Falls das SuchErgebnis nach diesem `commit` völlig leer sein sollte, weil alle Datensätze als gelöscht markiert waren und erfolgreicht gelöscht werden konnten, wird es automatisch geschlossen.
Falls der Cache des SuchErgebnisses nach diesem `commit` leer sein sollte, aber noch nicht alle Datensätze aus der Anfrage an die Datenbank gelesen worden sind, so wird ein Datensatz (der nächste) aus dem Anfrageergebnis gelesen. Dieser Lesevorgang ist die einzige Stelle in dieser Methode, an der eine SQLException geworfen werden kann.

+createCursor(): SuchErgebnisCursor

Erzeugt einen neuen Cursor zum Navigieren in diesem SuchErgebnis.

+forceEndReached(): void

Zwingt das SuchErgebnis in den Zustand „EndReached“. Das bedeutet, dass keine weiteren Dokumentdaten von der Datenbank mehr gelesen werden. Das SuchErgebnis verhält sich so, als ob bereits alle Daten aus der Datenbank gelesen worden wären.

+getCommitErrors(): Vector

Liefert einen Vektor mit allen Fehlern, die beim letzten `commit` dieses SuchErgebnisses aufgetreten sind. Liefert `Null` falls keine Fehler aufgetreten sind. Die Elemente des zurückgegebenen Vektors sind wieder Vektoren! Jeder dieser Vektoren enthält 2 oder 3 Elemente: Zuerst einen String (entweder „Dokument“ oder „commit“), dann die eigentliche SQLException, und bei den Dokumenten dann noch einen String mit dem Primärschlüssel des Dokuments. Es gibt höchstens einen „commit“-Eintrag, und wenn er existiert, befindet er sich immer am Ende des Vektors.

+getDokument (sec: SuchErgebnisCursor): DokumentDatensatz

Liefert den DokumentDatensatz an der aktuellen Position des SuchErgebnisCursors *sec*. Falls der Datensatz noch nicht aus dem ResultSet gelesen war, wird das jetzt gemacht. Falls der Cursor hinter dem Ende des SuchErgebnisses steht, wird er stillschweigend zurück an das Ende bewegt und das letzte Element des SuchErgebisses zurückgegeben. Falls das SuchErgebnis bereits geschlossen ist, wird `Null` zurückgegeben.

+getStatistics(): int[]

Liefert die Anzahl der DokumentDatensätze dieses SuchErgebnisses, die modifiziert (aber nicht als gelöscht markiert)/gelöscht/exklusiv sind. Der Rückgabewert ist ein int[3]-Array mit den Einträgen in der genannten Reihenfolge.

+isClosed(): boolean

Liefert `True`, falls das SuchErgebnis bereits geschlossen wurde.

+restore(): void

Setzt das SuchErgebnis in den Originalzustand zurück und verwirft alle daran vorgenommenen Modifikationen. Für jedes Dokument im SuchErgebnis, das bereits gelesen wurde, wird die Methode `restore` des DokumentDatensatzes aufgerufen, und er wird als „nicht gelöscht“ markiert. Hierfür sind keine Datenbankzugriffe nötig. Danach wird ein Datenbank-Rollback ausgeführt, um eventuell vorhandene Sperren wieder freizugeben. Alle DokumentDatensätze sind danach wieder im nicht-exklusiven Zustand.

+size(): int

Liefert die Anzahl der Datensätze des SuchErgebnisses, falls sie bereits bekannt ist. Ist sie noch nicht bekannt, so wird ein negativer Wert zurückgegeben, dessen Betrag eine Mindestanzahl darstellt. Falls das SuchErgebnis bereits geschlossen ist, wird 0 zurückgegeben. Die (Mindest)anzahl ist gleichzeitig die Anzahl der DokumentDatensätze, die schon im SuchErgebnis gecached sind.

7.51 Klasse SuchErgebnisCursor

extends java.lang.Object

Der SuchErgebnisCursor (Verzeichnis `shared/dok/`) ist ein Cursor zum Navigieren in einem SuchErgebnis. Man kann die Position des Cursors mit den geeigneten Methoden verschieben und dann aus dem SuchErgebnis den DokumentDatensatz an der Cursorposition holen.

Konstruktoren:

+SuchErgebnisCursor()

Erzeugt einen neuen SuchErgebnisCursor.

Methoden:

+getPos(): int

Liefert die Position dieses Cursors.

+movePos (rel: int): void

Bewegt die Position des Cursors um *rel* Stellen, wobei auch negative Werte erlaubt sind. Der Cursor wird auf einen Wert größer oder gleich 0 gezwungen.

+setPos (pos: int): void

Setzt die Position dieses Cursors. Der Cursor wird auf einen Wert größer oder gleich 0 gezwungen.

+setToFirstPos(): void

Setzt den Cursor auf den Anfang.

7.52 Klasse TextColArea

extends TextColItem

implements TextListener

Die TextColArea (Verzeichnis `shared/ui/`) erlaubt es, einen mehrzeiligen Text im Maskenfenster anzuzeigen und zu modifizieren. Eine TextColArea enthält eine FlitTextArea und einen FlitStrich.

Konstruktoren:

+TextColArea (name: String, uiname: String, rows: int, cols: int)

Erzeugt eine TextColArea mit dem internen Namen *name*, dem UINamen *uiname* und der gewünschten Anzahl an Zeilen *rows* und Spalten *cols*.

Methoden:

+getArea(): FlitTextArea

Liefert die FlitTextArea dieser TextColArea. Diese Funktion wird nur dazu benötigt, die FlitTextArea beim Konstruieren der Benutzeroberfläche in einen Container einfügen zu können.

+getText(): String

Liefert den aktuellen Wert des Textes.

+requestFocus(): void

Setzt den Eingabe-Fokus in diese TextColArea.

+setCaretPosition (pos: int): void

Setzt die aktuelle Cursorposition in dieser TextArea.

+setEditable (editable: boolean): void

Setzt die TextColArea auf editierbar oder nicht editierbar je nach Wert von *editable*. Im nicht-editierbaren Fall wird der FlitStrich automatisch auf „unsichtbar“ gesetzt.

+setText (text: String): void

Setzt den Wert des Textes auf *text*.

+textValueChanged (te: TextEvent): void

Implementaiton des TextListener-Interfaces, um die registrierten TextColListener benachrichtigen zu können.

7.53 Klasse TextColChoice

extends TextColItem

implements ItemListener

Die Klasse TextColChoice (Verzeichnis `shared/ui/`) erlaubt es, einen Text im Maskenfenster anzuzeigen. Der Text kann modifiziert werden, allerdings ist nur eine Auswahl fester Werte aus einer Liste von Alternativen möglich. Ein TextColField enthält einen FlitStrich.

Attribute:

-value: String

Der momentan ausgewählte Wert.

-items: Vector

Der Vektor mit den auswählbaren Einträgen.

-extravalue: String

Ein möglicherweise vorhandener zusätzlicher Wert.

Konstruktoren:

+TextColChoice (name: String, uiname: String, items: Vector)

Erzeugt eine TextColChoice mit dem internen Namen *name* und dem Vektor *items* als Alternativenliste. Eine Alternative selbst ist wiederum ein Vektor, der zwei Strings enthält: Den in der Benutzeroberfläche anzuzeigenden Text und den internen Text dieser Alternative.

Methoden:

+allowText (uitext: String, value: String): void

Fügt eine Alternative zum Vektor der Alternativen hinzu. Diese Alternative kann ab jetzt vom Benutzer ausgewählt werden. Falls bereits eine Alternative mit diesem *uitext* vorhanden ist, wird dieser Methodenaufruf ignoriert.

+forbidText (uitext: String): void

Entfernt eine Alternative. Sie kann ab jetzt nicht mehr vom Benutzer ausgewählt werden. Falls die Alternative gar nicht vorhanden war, wird dieser Methodenaufruf ignoriert.

+getChoice(): Choice

Liefert die Choice dieses TextColItems. Diese Funktion wird nur dazu benötigt, die Auswahlliste beim Konstruieren der Benutzeroberfläche in einen Container einfügen zu können.

+getText(): String

Liefert den aktuellen Wert des Textes.

+itemStateChanged (e: ItemEvent): void

Implementation des ItemListener-Interfaces, um die registrierten TextColListener benachrichtigen zu können.

+requestFocus(): void

Setzt den Eingabe-Fokus in diese TextColChoice.

+setEditable (editable: boolean): void

Setzt die TextColChoice auf editierbar oder nicht editierbar je nach Wert von *editable*. Im nicht-editierbaren Fall wird der FlitStrich automatisch auf „unsichtbar“ gesetzt.

+setText (text: String): void

Setzt den Wert des Textes auf *text*. Falls dieser Wert nicht im Vektor der Alternativen vorgesehen ist, wird er als „Ausnahme-Alternative“ hinzugefügt und dann auch angezeigt. Eine möglicherweise bereits vorhandene „Ausnahme-Alternative“ wird in jedem Fall wieder entfernt.

+setValueForUIText (value: String, uitext: String): void

Ändert den internen Text einer vorhandenen Alternative, die über den angezeigten Text *uitext* identifiziert wird, auf *value*. Wenn keine Alternative mit dem angegebenen Text vorhanden ist, ist dieser Aufruf funktionslos.

7.54 Klasse TextColField

extends TextColItem

implements TextListener

Ein TextColField (Verzeichnis `shared/ui/`) erlaubt es, einen Text im Maskenfenster anzuzeigen und zu modifizieren. Es enthält ein FlitTextField und einen FlitStrich. Die Feldgröße kann auf eine Mindestgröße eingestellt werden.

Konstruktoren:

+TextColField (name: String, uiname: String, width: int)

Erzeugt ein TextColField mit dem internen Namen *name* und *width* sichtbaren Zeichen.

Methoden:

+getField(): FlitTextField

Liefert das FlitTextField dieses TextColFields. Diese Funktion wird nur dazu benötigt, das FlitTextField beim Konstruieren der Benutzeroberfläche in einen Container einfügen zu können.

+getText(): String

Liefert den aktuellen Wert des Textes.

+requestFocus(): void

Setzt den Eingabe-Fokus in dieses TextColField.

+setCaretPosition (pos: int): void

Setzt die Cursorposition in diesem Textfeld.

+setEditable (editable: boolean): void

Setzt das TextColField auf editierbar oder nicht editierbar je nach Wert von *editable*. Im nicht editierbaren Fall wird der FlitStrich automatisch auf „unsichtbar“ gesetzt.

+setText (text: String): void

Setzt den Text auf *text*.

+textValueChanged (te: TextEvent): void

Implementation des TextListener-Interfaces, um die registrierten TextColListener benachrichtigen zu können.

7.55 Klasse TextColFieldwithList

extends TextColField

Die Klasse TextColFieldwithList (Verzeichnis `shared/ui/`) erweitert die Klasse TextColField um die Möglichkeit, die bisher vorhandenen Werte aus der Datenbank auszulesen und zur Auswahl anzubieten.

Konstruktoren:

+TextColFieldWithList (name: String, uiname: String, width: int)

Erzeugt ein TextColField mit dem internen Namen *name* und *width* sichtbaren Zeichen, das einen Knopf zum öffnen eines Listenfensters besitzt.

Methoden:

+setEditable (editable: boolean): void

Setzt das TextColField und den zugehörigen Knopf auf editierbar oder nicht editierbar je nach Wert von *editable*. Im nicht editierbaren Fall wird der FlitStrich automatisch auf „unsichtbar“ gesetzt.

+getButton(): Button

Liefert den Button. Diese Funktion wird nur dazu benötigt, den Button beim Konstruieren der Benutzeroberfläche in einen Container einfügen zu können.

7.56 Klasse TextColItem

extends java.lang.Object

implements FocusListener

Die abstrakte Klasse TextColItem (Verzeichnis `shared/ui/`) ist die Oberklasse aller Klassen, die es erlauben, einen Text im Maskenfenster anzuzeigen und zu modifizieren. Sie enthält neben der eigentlichen Komponente zum Editieren des Textes einen FlitStrich zur Anzeige von Statusinformationen. Die Farbe des Striches kann gesetzt werden. Der Text selbst kann gesetzt und ausgelesen werden. Um über Veränderungen des Textes informiert zu werden, kann man einen TextColListener bei diesem TextColItem registrieren lassen.

Attribute:

#name: String

Der interne Name dieses TextColItems.

#uiname: String

Der Name dieses TextColItems, mit dem der Benutzer dieses TextColItem identifizieren kann.

#strich: FlitStrich

Der FlitStrich, der den Status dieses TextColItems anzeigt.

#editable: boolean

Gibt an, ob dieses TextColItem editierbar ist.

#listener: TextColListener

Der registrierte TextColListener für dieses TextColItem.

Methoden:

+allowText (uitext: String, value: String): void

Erlaubt einen Textwert. Diese Methode bekommt in der Unterklasse TextColChoice eine Funktion.

+focusGained (e: FocusEvent): void

Implementation des FocusListener-Interfaces.

+focusLost (e: FocusEvent): void

Implementation des FocusListener-Interfaces.

+forbidText (uitext: String): void

Verbietet einen Textwert. Diese Methode bekommt in der Unterklasse TextColChoice eine Funktion.

+getStrich(): FlitStrich

Liefert den FlitStrich dieses TextColItems. Diese Funktion wird ausschließlich dazu benötigt, den FlitStrich beim Konstruieren der Benutzeroberfläche in einen Container einfügen zu können.

+getText(): String

Liefert den aktuellen Wert des Textes.

+getUIName(): String

Liefert den UINamen dieses TextColItems. Es ist ein String ohne führende oder abschließende Leerzeichen, anhand dessen der Benutzer dieses Feld identifizieren kann.

+isEditable(): boolean

Gibt an, ob das TextColItem editierbar ist.

+requestFocus(): void

Setzt den Eingabe-Fokus in dieses TextColItem. Diese Methode bekommt erst in den Unterklassen eine Funktion.

+setCaretPosition (pos: int): void

Setzt die Cursorposition in diesem TextColItem. Diese Methode bekommt in der Unterklasse TextColField eine Funktion.

+setColorMode (i: int): void

Setzt die Farbe des FlitStriches auf *i*.

+setEditable (editable: boolean): void

Setzt das TextColItem auf editierbar oder nicht editierbar je nach Wert von *editable*.

+setFlitFocusListener (flf: FlitFocusListener): void

Setzt einen FlitFocusListener für dieses TextColItem. Es ist nur ein einziger FlitFocusListener pro TextColItem vorgesehen, d. h. nur der zuletzt gesetzte FlitFocusListener wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein FlitFocusListener mehr registriert.

+setText (text: String): void

Setzt den Text auf den Wert *text*.

+setTextColListener (tl: TextColListener): void

Setzt einen TextColListener für dieses TextColItem. Es ist nur ein einziger TextColListener pro TextColItem vorgesehen, d. h. nur der zuletzt gesetzte TextColListener wird benachrichtigt. Darf mit `Null` aufgerufen werden; in diesem Fall ist dann kein TextColListener mehr registriert.

7.57 Interface TextColListener

Der TextColListener (Verzeichnis `shared/ui/`) ist ein abstraktes Interface, ähnlich dem TextListener. Der Unterschied ist, dass bei der Änderung des Textes nicht wie beim TextListener ein TextEvent, sondern einfach die Quelle, ihr Name und der neue Text übergeben werden.

Methoden:

+textColValueChanged (source: TextColItem, name: String, text: String): void

Wird aufgerufen, wenn sich der Wert eines Textes geändert hat. Dabei ist *source* das TextColItem mit dem Namen *name*, in dem das Event ausgelöst wurde, und *text* ist der neue Wert des Textes.

Kapitel 8

Besonderheiten in der Implementation von *Flit*

Die Maske *Flit* ist das eigentliche Original der Maskenfamilie. In diesem Sinne sind die hier dokumentierten Klassen keine Erweiterungen der *shared*- bzw. *template*-Klassen, sondern andersherum jene eine Reduktion dieser. Allerdings ist das eher ein Detail, auf das hier nicht weiter eingegangen werden soll.

Die Anforderungen an die Maske *Flit* unterschieden sich von den Grundanforderungen in der Art, dass die Dokumente in der Datenbankrelation in anderen Dokumenten enthalten sein können. Dies erforderte die zusätzliche Klasse DokumentIndok und entsprechende Erweiterungen an der Klasse DokumentDatensatz.

Darüber hinaus werden zu den einzelnen Dokumenten noch Schlagwörter in einer separaten Datenbanktabelle verwaltet. Die Erweiterungen am SQL-Assistenten und die für die Schlagwörter eingeführten neuen *dok*- und *ui*-Klassen tragen dem Rechnung.

8.1 Klasse DokumentIndok

extends java.lang.Objekt

Die Klasse DokumentIndok (Verzeichnis `flit/dok/`) enthält einige Felder des „INDOK“-Dokuments. Jedes Feld wird durch einen String identifiziert; String-Konstanten für alle Feldnamen stehen statisch in dieser Klasse bereit. Der Wert eines Feldes ist ebenfalls ein String, der nicht „NULL“ sein darf. Zur Speicherung eines Datenbank-NULL-Wertes wird hier der leere String vereinbart.

Attribute:

+$DOKTYP: String

Ein String mit dem Namen des Feldes „DOKTYP“.

+$AUTOREN: String

Ein String mit dem Namen des Feldes „AUTOREN“.

+$TITEL: String

Ein String mit dem Namen des Feldes „TITEL“.

+$fieldnames: String[]

Ein Array mit den Namen aller INDOK-Felder.

Konstruktoren:

+DokumentIndok()

Erzeugt ein neues, leeres DokumentIndok.

Methoden:

+get (feld: String): String

Liefert den Wert des Feldes *feld*. Falls der Wert nicht gesetzt wurde, wird der leere String zurückgegeben.

#set (feld: String, wert: String): void

Setzt den Wert des Feldes *feld* auf *wert*. Darf mit 'wert = null' aufgerufen werden; in diesem Fall wird der leere String gesetzt.

8.2 Klasse DokumentSchlagwoerter

extends java.lang.Objekt

Eine Instanz der Klasse DokumentSchlagwoerter (Verzeichnis `flit/dok/`) ist eine Sammlung aller Schlagwörter eines Dokuments. Diese Instanz kann als „modifizierbar“ und als in der Datenbank „exclusive“ markiert werden. Um auf alle Schlagwörter in dieser Sammlung zugreifen zu können, sind sie durchnummeriert.

Jedes Schlagwort selbst kann aus der Datenbank gelesen (dann hat es eine ROWID und ist entweder unmodifiziert, modifiziert oder gelöscht) oder neu eingegeben sein (dann hat es keine ROWID).

Attribute:

-schlag: Vector

Der Vektor, der die einzelnen Schlagwörter speichert.

-dok: DokumentDatensatz

Der DokumentDatensatz, zu dem diese Schlagwortsammlung gehört.

Konstruktoren:

+DokumentSchlagwoerter (modifiable: boolean)

Erzeugt neue leere DokumentSchlagwoerter, die je nach Wert von *modifiable* modifizierbar sind oder nicht.

Methoden:

+add (schlagwort: String, kuerzel: String, gewicht: int, rowid: String): void

Fügt ein Schlagwort mit den angegebenen Parametern und ROWID zur Sammlung hinzu.

+add (schlagwort: String, kuerzel: String, gewicht: int): void

Fügt ein Schlagwort mit den angegebenen Parametern, allerdings ohne ROWID, zur Sammlung hinzu. Falls die DokumentSchlagwoerter nicht modifizierbar sind, wird dieser Aufruf ignoriert.

#commit(): boolean

Übernimmt alle Modifikationen als neuen Originalzustand. Falls es Schlagwörter ohne ROWID in der Sammlung gibt, schlägt dieser Aufruf fehl und gibt `False` zurück. Man muss dann diese Instanz DokumentSchlagwoerter verwerfen und eine neue erzeugen.

+contains (schlagwort: String, kuerzel: String, gewicht: int, rowid: String): boolean

Testet, ob sich ein Schlagwort mit den übergebenen Parametern in der Sammlung befindet.

+delete (nr: int): void

Löscht das Schlagwort an der Position *nr*; die Zählung beginnt bei 0. Hat das Schlagwort eine ROWID, so wird es nur als gelöscht markiert, bleibt aber in der Sammlung enthalten. Hat es keine ROWID, wird es aus der Sammlung entfernt. Wirf ggf. eine ArrayIndexOutOfBoundsException.

+delete (s: Schlagwort): void

Löscht das Schlagwort *s* aus der Sammlung. Hat das Schlagwort eine ROWID, so wird es nur als gelöscht markiert, bleibt aber in der Sammlung enthalten. Hat es keine ROWID, wird es aus der Sammlung entfernt. Es ist dabei irrelevant, ob es überhaupt in der Sammlung enthalten war.

+get (nr: int): Schlagwort

Liefert das Schlagwort an der Position *nr*; die Zählung beginnt bei 0. Wirft ggf. eine ArrayIndexOutOfBoundsException.

+getDokument(): DokumentDatensatz

Liefert den DokumentDatensatz, zu dem diese DokumentSchlagwoerter gehören.

+getKurzuebersicht(): String

Liefert eine Kurzübersicht über die Schlagwörter. Diese besteht aus einer durch Kommata getrennten Liste der ersten zehn Schlagwörter und dem jeweiligen Gewicht in Klammern hinter jedem Schlagwort.

#getOriginalSize(): int

Liefert die Anzahl der Schlagwörter in der Sammlung, die eine ROWID besitzen.

+indexOf (s: Schlagwort): int

Liefert die Nummer des Schlagwortes *s*; die Zählung beginnt bei 0. Falls das Schlagwort nicht in der Sammlung ist, wird -1 zurückgegeben.

+isExclusive(): boolean

Gibt an, ob diese DokumentSchlagwoerter in der Datenbank zum exklusiven Zugriff gesperrt wurden.

+isModifiable(): boolean

Gibt ab, ob diese DokumentSchlagwoerter modifizierbar sind.

+isModified(): boolean

Gibt an, ob die DokumentSchlagwoerter modifiziert wurden, d. h. ob irgendein Schlagwort der Sammlung modifiziert oder als gelöscht markiert wurde.

+restore(): void

Löscht alle Modifikationen und stellt den Originalzustand wieder her. Alle Schlagwörter ohne ROWID werden entfernt.

+restore (nr: int): void

Stellt den Originalzustand des Schlagwortes an der Position *nr* wieder her; die Zählung beginnt bei 0. Hat das Schlagwort keine ROWID, ist dieser Aufruf funktionslos. Wirft ggf. eine ArrayIndexOutOfBoundsException.

#setDokument (DokumentDatensatz dok): void

Teilt diesen DokumentSchlagwoertern mit, zu welchem DokumentDatensatz sie gehören.

+setExclusive (exclusive: boolean): void

Markiert die DokumentSchlagwoerter als exklusiv oder nicht exklusiv je nach Wert von *exclusive*.

+setModifiable (modifiable: boolean): void

Markiert die DokumentSchlagwoerter als modifizierbar oder nicht modifizierbar je nach Wert von *modifiable*. Falls die DokumentSchlagwoerter als „nicht modifizierbar“ markiert werden, wird der Originalzustand aller Schlagwörter wiederhergestellt (indem die Modifikationen verworfen werden), und alle neu eingegebenenn Schlagwörter (die ohne ROWID) werden entfernt.

+size(): int

Liefert die Anzahl der Schlagwörter in der Sammlung.

8.3 Klasse ExprSchlagwortParser

extends ExprParser

Der ExprSchlagwortParser (Verzeichnis `flit/dok/`) ist eine Erweiterung des ExprParsers wie die anderen Erweiterungen im Verzeichnis `shared/dok/` auch; geparst werden hier Strings auf Items, die Schlagwörter mit eventueller Gewichtsbedingung sind.

Aufrufer parsen einen String mit der Methode `parse` und holen sich, falls kein Fehler beim Parsen aufgetreten ist, danach den in SQL übersetzten String mit `getSQL`. Diese Methoden finden sich in der Klasse EXPRPARSER.

Vor dem ersten Parsen muss der SQL-Spaltenname auf „SCHLAGWORT“ und der SQL-Spaltenname der Schlagwortgewichte mit `setGewichtSpalte` auf „GEWICHT“ gesetzt werden. Es darf kein Tabellenbezeichner (wie etwa „s.“) vorangestellt werden.

Konstruktoren:

+ExprSchlagwortParser()

Default-Konstruktor; leer.

Methoden:

#parseItem(): boolean

Parst ein Item, das bei dieser Implementation ein Schlagwort mit eventueller Gewichtsbedingung ist. Das Schlagwort wird durch eines der Zeichen „)“, „&“, „|“ oder das Stringende terminiert. Ausnahme: zu jeder im Schlagwort auftauchenden öffnenden Klammer „(“ übernimmt diese Implementation auch eine eventuell auftauchende schließende Klammer „)“ mit in das Schlagwort oder in die Gewichtsbedingung, ohne es zu terminieren. Gibt `True` zurück, falls das Parsen fehlerfrei durchgeführt werden konnt.

+setGewichtSpalte (spalteGewicht: String): void

Setzt den SQL-Spaltennamen der Spalte der Schlagwortgewichte, üblicherweise „GEWICHT“. Es darf kein Tabellenbezeichner (wie etwa „s.“) vorangestellt werden, weil das diese Methode selbst macht.

8.4 Klasse SCHLAGWORT

extends java.lang.Object

Eine Instanz der Klasse SCHLAGWORT (Verzeichnis `flit/dok/`) besteht aus dem eigentlichen Schlagwort, einem Namenskürzel, einem Gewicht und einer ROWID.

Konstruktoren:

#Schlagwort (schlagwort: String, kuerzel: String, gewicht: int)

Erzeugt ein neues SCHLAGWORT ohne ROWID. Intern wird ROWID = `Null` gespeichert.

#Schlagwort (schlagwort: String, kuerzel: String, gewicht: int, rowid: String)

Erzeugt ein SCHLAGWORT mit ROWID.

Methoden:

#commit(): boolean

Übernimmt alle Modifikationen als neuen Originalzustand. Ist die ROWID gleich `Null`, so schlägt dieser Aufruf fehl und gibt `False` zurück, ansonsten `True`.

#delete(): void

Markiert das SCHLAGWORT als gelöscht.

+getGewicht(): int

Holt das Gewicht des SCHLAGWORTES.

+getKuerzel(): String

Holt das Kürzel.

+getROWID(): String

Hold die ROWID; gibt `Null` zurück, falls beim Erzeugen keine ROWID angegeben wurde.

+getSchlagwort(): String

Holt den Schlagwort-String.

+hasModifiedGewicht(): boolean

Gibt an, ob das Gewicht des SCHLAGWORTES modifiziert wurde.

+hasModifiedSchlagwort(): boolean

Gibt an, ob der Schlagwort-String modifiziert wurde.

+isDeleted(): boolean

Gibt an, ob das SCHLAGWORT als gelöscht markiert wurde.

+isModified(): boolean

Gibt an, ob das SCHLAGWORT (im Schlagwort-String, Kürzel oder Gewicht) verändert wurde, oder die ROWID gleich `Null` ist.

+restore(): boolean

Ist die ROWID gleich `Null`, wird dieser Aufruf ignoriert; ansonsten löscht er alle Modifikationen und stellt den Originalzustand wieder her. Das SCHLAGWORT wird dann als „nicht gelöscht“ markiert.

+setGewicht (gewicht: int): void

Setzt das Gewicht des SCHLAGWORTES auf *gewicht*. Das SCHLAGWORT ist damit modifiziert, falls der Wert ungleich dem Originalwert ist.

+setKuerzel (kuerzel: String): void

Setzt das Kürzel des SCHLAGWORTES auf *kuerzel*. Das SCHLAGWORT ist damit modifiziert, falls der Wert ungleich dem Originalwert ist.

+setSchlagwort (schlagwort: String): void

Setzt den Schlagwort-String auf *schlagwort*. Das SCHLAGWORT ist damit modifiziert, falls der Wert ungleich dem Originalwert ist.

#undelete(): void

Markiert das Schlagwort als nicht gelöscht. Die Feldinhalte des Schlagwortes bleiben unverändert.

#wasOriginal (schlagwort: String, kuerzel: String, gewicht: String): boolean

Gibt `True` zurück, falls die übergebenen Daten mit den Originaldaten des Schlagwortes übereinstimmen.

8.5 Klasse SchlagwortCanvas

extends java.awt.Canvas

implements MouseListener, AdjustmentListener

Der SchlagwortCanvas (Verzeichnis `flit/ui/`) ist die Zeichenfläche eines SchlagwortFrames, die die Schlagwörter eines Dokuments am Bildschirm in Listenform darstellt.

Konstruktoren:

+SchlagwortCanvas (font: Font, sf: SchlagwortFrame)

Erzeugt einen neuen SchlagwortCanvas für den SchlagwortFrame *sf*. Dieser enthält intern einen TitelCanvas und eine Canvas-Scrollbar.

Methoden:

+adjustmentValueChanged (e: AdjustmentEvent): void

Implementation des AdjustmentListener-Interfaces.

+getCanvasScrollbar(): Scrollbar

Holt die CanvasScrollbar dieses SchlagwortCanvasses. Diese Methode wird nur für Layout-Zwecke benötigt.

+getMaximumSize(): Dimenstion

Liefert die maximale Ausdehnung dieses SchlagwortCanvasses.

+getMinimumSize(): Dimension

Liefert die minimale Ausdehnung dieses SchlagwortCanvasses.

+getPreferredSize(): Dimension

Liefert die gewünschte Ausdehnung dieses SchlagwortCanvasses.

+getTitelCanvas(): Canvas

Holt die TitelCanvas dieses SchlagwortCanvasses. Diese Methode wird nur für Layout-Zwecke benötigt.

+mouseClicked (e: MouseEvent): void

Implementation des MouseListener-Interfaces; leer.

+mouseEntered (e: MouseEvent): void

Implementation des MouseListener-Interfaces; leer.

+mouseExited (e: MouseEvent): void

Implementation des MouseListener-Interfaces; leer.

+mousePressed (e: MouseEvent): void

Implementation des MouseListener-Interfaces; setzt die Selektion.

+mousReleased (e: MouseEvent): void

Implementation des MouseListener-Interfaces; leer.

+paint (g: Graphics): void

Zeichnet den SchlagwortCanvas.

+setSchlagwoerter (d: DokumentSchlagwoerter): void

Zeigt die DokumentSchlagwoerter *d* an. Der Wert `Null` ist erlaubt. Die Selektion und die Scrollposition werden auf 0 gesetzt. Falls jedoch `Null` übergeben wurde oder die DokumentSchlagwoerter nicht modifizierbar oder leer sind oder der zugehörige DokumentDatensatz als gelöscht markiert ist, wird die Selektion auf -1 gesetzt.

+setSelection (n: int): void

Setzt die Selektion auf eine bestimmte Zeile. Die Scrollposition wird so angepasst, dass die Selektion im sichtbaren Bereicht liegt. Der Wert -1 bedeutet keine Selektion.

+update (g: Graphics): void

Zeichnet den SchlagwortCanvas.

8.6 Klasse SchlagwortFrame

extends java.awt.Frame

implements ActionListener, KeyListener, TextColListener

Der SchlagwortFrame (Verzeichnis `flit/ui/`) ist das Schlagwortfenster der *Flit*-Maske. Es enthält einen SchlagwortCanvas sowie Felder zum Eingeben und Ändern von Schlagwörtern und deren Gewicht, ein Menü und Statuslabel.

Konstruktoren:

+SchlagwortFrame (font: Font)

Erzeugt einen neuen SchlagwortFrame, der vom Aufrufer noch mit `setVisible` angezeigt werden muss.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+detach(): void

Löst diesen SchlagwortFrame von allen Referenzen auf SchlagwortListener und DokumentSchlagwoerter.

+displayAndToFront(): void

Macht diesen SchlagwortFrame sichtbar, bringt ihn nach vorne und setzt den Fokus in ein Textfeld.

+displaySchlagwoerter (d: DokumentSchlagwoerter): void

Zeigt die DokumentSchlagwoerter *d* an. Der Wert `Null` ist erlaubt. Die Selektion wird auf 0 gesetzt. Falls jedoch `Null` übergeben wurde oder die DokumentSchlagwoerter nicht modifizierbar oder leer sind oder der zugehörige DokumentDatensatz als gelöscht markiert ist, wird die Selektion auf -1 gesetzt.

+getSelection(): int

Liefert die Selektion dieses SchlagwortFrames.

+keyPressed (e: KeyEvent): void

Implementation des KeyListener-Interfaces.

+keyReleased (e: KeyEvent): void

Implementation des KeyListener-Interfaces.

+keyTyped (e: KeyEvent): void

Implementation des KeyListener-Interfaces.

+pleasePositionNextTo (parent: Frame): void

Positioniert den SchlagwortFrame so auf dem Bildschirm, dass er möglichst neben dem *parent*-Frame erscheint, ohne sich jedoch über den Bildschirmrand hinaus zu erstrecken.

+selection_callback (n: int): void

Diese Methode wird vom SchlagwortCanvas aufgerufen, wenn sich die Selektion durch Benutzeraktionen im Canvas (d. h. Mausklicks, Scrollbar) geändert hat.

+selectLastLint(): void

Selektiert die letzte Zeile der DokumentSchlagwoerter und setzt den Fokus in das Eingabefeld.

+setFieldsEditable (editable: boolean): void

Setzt die Eingabefelder des SchlagwortFrames auf editierbar oder nicht editierbar je nach Wert von *editable*.

+setFieldValues (schlagwort: String, col: int, gewicht: String, colg: int): void

Setzt die Werte der Eingabefelder des SchlagwortFrames. Dabei ist *schlagwort* der String für das Feld „Schlagwort“, das den Farbmoduswert *col* bekommt, und *gewicht* der String für das Feld „Gewicht“, das den Farbmoduswert *colg* bekommt.

+setKeyListener (kl: KeyListener): void

Setzt den KeyListener für diesen SchlagwortFrame. Der Wert `Null` ist erlaubt; dann ist kein KeyListener mehr registriert.

+setLabel (s: String): void

Setzt die „Fehler"-Statuszeile des SchlagwortFrames auf den String *s*. Wird `Null` oder der leere String übergeben, wird eine Information über die Schlagwörter angezeigt (unverändert/gelöscht/nicht änderbar/exklusiv)

+setSchlagwortListener (sl: SchlagwortListener): void

Setzt den SchlagwortListener für diesen SchlagwortFrame; der Wert `Null` ist erlaubt, dann ist kein SchlagwortListener mehr registriert.

+textColValueChanged (tci: TextColItem, name: String, text: String): void

Implementation des TextColListener-Interfaces.

+updateAll(): void

Aktualisiert alle Daten im Fenster; wird nur benötigt, nachdem eine Zeile gelöscht wurde oder nachdem alle Änderungen rückgängig gemacht wurden.

+updateSelectedLine(): void

Aktualisiert die Daten der selektierten Zeile im Fenster.

8.7 Interface SchlagwortListener

Der SchlagwortListener (Verzeichnis `flit/ui/`) ist ein Interface, das zur Kommunikation zwischen dem SchlagwortFrame und dem Arbeitsablauf dient.

Methoden:

+schlagwort_executeCommand (ac: String): void

Wird aufgerufen, wenn der Befehl mit dem *ActionCommand ac* ausgeführt werden soll.

+schlagwort_selectionChanged (n: int): void

Wird aufgerufen, wenn sich die Selektion (= Nummer der selektierten Zeile) geändert hat.

+schlagwort_textColValueChanged (source: TextColItem, name: String, text: String): void

Wird aufgerufen, wenn sich der Wert des Eingabefeldes *source* mit dem Namen *name* geändert hat. Der neue Inhalt ist *text*.

8.8 Klasse ARBEITSABLAUF (Erweitert)

Diese Klasse muss selbstverständlich die erweiterten Funktionen für die enthaltenden Dokumente und die Schlagwörter zur Verfügung stellen. Dazu sind natürlich alle bereits in der Template-Klasse enthaltenen Funktionen entsprechend erweitert worden; das soll hier allerdings nicht genauer ausgeführt werden. Stattdessen werden hier nur die Methoden genannt, die neu hinzugekommen sind.

Außerdem wurde in einer neuen privaten Variable abgelegt, bei welchen Dokumenttypen welche Felder zusätzlich deaktiviert sein sollen.

Attribute:

-dok_doktyp2gesperrteFelder: Hashtable

Enthält in der Form (DOKTYP $\mapsto$ Vector) diejenigen Eingabefelder, die bei einem bestimmten DokTyp deaktiviert sein sollen. Die Namen der zu sperrenden TEXTCOLITEMS sind im jeweiligen Vektor zu finden.

Methoden:

+schlagwort_executeCommand (ac: String): void

Implementation des SCHLAGWORTLISTENER-Interfaces.

+schlagwort_selectionChanged (n: int): void

Implementation des SCHLAGWORTLISTENER-Interfaces.

+schlagwort_textColValueChanged (tci: TEXTCOLITEM, name: String, text: String): void

Implementation des SCHLAGWORTLISTENER-Interfaces.

-tryToLockSchlagwoerter (ds: DOKUMENTSCHLAGWOERTER): String

Versucht, die DOKUMENTSCHLAGWOERTER *ds* in der Datenbank zu sperren. Liefert `Null` im Erfolgsfall, andernfalls einen Fehlerstring.

8.9 Klasse DOKUMENTDATENSATZ (Erweitert)

Der DOKUMENTDATENSATZ wurde um die Felder des enthaltenden Dokumentes erweitert. Außerdem wurden sowohl dafür als auch für die Zugriffe auf Schlagwörter, die zu diesem Dokument gehören, die folgenden neuen Methoden notwendig:

Methoden:

+setSchlagwoerter (s: DOKUMENTSCHLAGWOERTER): void

Verknüpft die DOKUMENTSCHLAGWOERTER *s* mit diesem DOKUMENTDATENSATZ. Dabei wird auch bei den DOKUMENTSCHLAGWOERTERN die Methode „setDokument“ aufgerufen. Darf mit dem Wert `Null` aufgerufen werden; in diesem Fall wurden die (aktuellen) Schlagwörter noch nicht aus der Datenbank geholt.

+getSchlagwoerter(): DokumentSchlagwoerter

Liefert die DokumentSchlagwoerter zu diesem DokumentDatensatz. Falls die Schlagwörter noch ermittelt werden müssen, wird `Null` zurückgegeben.

+setIndok (i: DokumentIndok): void

Verknüpft das DokumentIndok *i* mit diesem DokumentDatensatz. Darf mit dem Wert `Null` aufgerufen werden; in diesem Fall wurden die (aktuellen) Indok-Daten noch nicht aus der Datenbank geholt.

+getIndok(): DokumentIndok

Liefert das DokumentIndok zu diesem DokumentDatensatz. Falls die Indok-Daten noch ermittelt werden müssen, wird `Null` zurückgegeben.

+restore(): void

Löscht alle Modifikationen und stellt den Originalzustand wieder her. Die DokumentSchlagwoerter bleiben hiervon unberührt!

8.10 Klasse SQLAssistant (Erweitert)

Der SQLAssistant unterscheidet sich von der Klasse der automatischen Instantiierung dadurch, dass er zusätzliche Methoden für das Auslesen der Schlagwörter anbietet. Die Daten selbst werden aus der Datenbankrelation `schlagwort` ausgelesen.

Methoden:

+createDokumentIndok (doknr: String): DokumentIndok

Holt zu einem Dokument das DokumentIndok mit der INDOK *doknr* aus der Datenbank.

-insertNewSchlagwort (doknr: String, s: Schlagwort): void

Fügt das neue Schlagwort *s* in die Datenbank ein, so dass es zu dem Dokument mit der Dokumentnummer *doknr* gehört.

-updateSchlagwort (s: Schlagwort): void

Aktualisiert das modifizierte Schlagwort *s* in der Datenbank.

-deleteSchlagwort (s: Schlagwort): void

Löscht das Schlagwort *s* aus der Datenbank.

+lockSchlagwoerter (ds: DokumentSchlagwoerter): int

Sperrt die DokumentSchlagwörter *ds* in der Datenbank für exklusiven Zugriff (select for update nowait) und prüft, ob die aktuellen Daten in der Datenbank noch mit den Originaldaten der DokumentSchlagwoerter übereinstimmen. Der zugehörige DokumentDatensatz muss bereits im exklusiven Zustand sein!
In Abhängigkeit vom Ausgang des Sperrversuchs und des Datenvergleichs werden die DokumentSchlagwoerter in einen bestimmten Zustand gesetzt und ein Integer-Wert zurückgegeben:

Faktenlage	DokumentSchlagwoerter	Wert
Sperren erfolgreich, Daten unverändert	exclusive	1
Sperren erfolgreich, Daten wurden verändert	exclusive, unmodifizierbar	0
Sperren fehlgeschlagen	unmodifizierbar	-1

Der Rückgabewert ist also genau dann <= 0, wenn die DokumentSchlagwoerter jetzt nicht mehr modifizierbar sind, und genau dann >= 0, wenn die DokumentSchlagwoerter jetzt exklusiv sind. Man beachte, dass im Fall = 0 die DokumentSchlagwörter zwar gesperrt, aber dennoch nicht modifzierbar sind.

8.11 Klasse SuchErgebnis (Modifiziert)

Die meisten Methoden des SuchErgebnisses sind intern derart erweitert worden, dass sie die Schlagwörter gleich mit verwalten. Die einzige Stelle, an der dieser Unterschied außerhalb bemerkbar ist, ist die Erweiterung der Methode `getStatistics` wie folgt:

Methoden:

+getStatistics(): int[]

Liefert die Anzahl der DokumentDatensätze in diesem SuchErgebnis, die modifiziert/gelöscht/exklusiv sind, und wie viele DokumentSchlagwoerter modifiziert/gelöscht sind. Der Rückgabewert ist ein Array, das wie folgt zu interpretieren ist:

Position	Wert: Anzahl der im SuchErgebnis...
0	modifizierten, aber nicht als gelöscht markierten Datensätze
1	als gelöscht markierten DokumentDatensätze
2	als exklusiv markierten DokumentDatensätze
3	modifizierten DokumentSchlagwoerter
4	als exklusiv markierten DokumentSchlagwoerter

8.12 ResourceBundle `Sprache.properties` (Erweitert)

Ein zusätzliche Funktion von *Flit* ist es, dass Maskenelemente in Abhängigkeit vom Dokumenttyp des momentan angezeigten Dokuments deaktiviert werden können.

Der ARBEITSABLAUF liest dazu aus dem ResourceBundle `Sprache.properties` zusätzlich Daten der Form

`dok.text.blocked.erg.`*`DOKTYP`*`.`*`X`*`=`*`TEXT`*

ein, wobei *`DOKTYP`* der Typ des angezeigten Dokuments ist, *`X`* eine fortlaufende Nummer für diesen Typ, und *`TEXT`* der Name des Maskenelementes ist, das zusätzlich deaktiviert werden soll.

Kapitel 9

Besonderheiten in der Implementation von *MetaMask*

9.1 eine andere Aufgabe

MetaMask ist keine Maske der *Flit*-Familie im eigentlichen Sinne, da sie auf keiner Datenbanktabelle basiert. Stattdessen muss *MetaMask* Files lesen und schreiben können und die Daten der anderen Masken verwalten. Aus diesem Grund unterscheiden sich die meisten Klassen im Verzeichnis `metamask/dok/` deutlich von den automatisch erzeugten Klassen.

Die Klassen, die zur Darstellung der Maske selbst benutzt werden, sind jedoch dieselben; bis auf die Ergänzung zweier Frames, die in den üblichen Masken nicht gebraucht werden.

Die folgenden Abschnitte sollen die einzelnen Klassen genauer erläutern.

9.2 Klasse Arbeitsablauf

extends java.lang.Object

Der Arbeitsablauf (Verzeichnis `metamask/dok/`) hat mit dem Arbeitsablauf einer automatisch erzeugten Maske außer der logischen Position eher wenig gemeinsam. Das liegt einerseits daran, dass *MetaMask* nur zwei statt drei Programmzustände kennt, aber andererseits auch daran, dass keine Datenbankzugriffe für die Modifikation der Daten nötig sind und andere Listener implementiert werden mussten.

Auf die dennoch vorhandenen Ähnlichkeiten dieser Klasse mit dem üblichen Arbeitsablauf soll hier nicht weiter eingegangen werden.

Die interne Klasse TestListener implementiert die notwendigen Listener für das Testfenster. Dadurch wird es möglich, den ARBEITSABLAUF auf ActionEvents des Testfensters reagieren zu lassen.

Attribute:

-dm: DATAMANAGER

Der DATAMANAGER, der die Daten der bearbeiteten Maske zur Verfügung stellt.

-fm: FileManager

Der FileManager, der die Zugriffe auf die Templates und die erzeugten Klassenfiles ermöglicht.

Konstruktoren:

+Arbeitsablauf (mf: FLITFRAME, dm: DATAMANAGER)

Erzeugt einen neuen ARBEITSABLAUF, der sich auf den MainFrame *mf* bezieht, und die Daten aus dem DATAMANAGER *dm* gewinnt.

Methoden:

+choiceCancel(): void

Implementation des CHOICEEDITLISTENER-Interfaces.

+choiceChangedText (newChoices: CHOICEDATA): void

Implementation des CHOICEEDITLISTENER-Interfaces.

+executeRepositionCommand (ac: String, row: int, col: int, ok: boolean): void

Implementation des REPOSITIONFRAMELISTENER-Interfaces.

+setNumber(ArbeitsablaufNummer: int): void

Setzt die Nummer des ARBEITSABLAUFES auf *ArbeitsablaufNummer*. Dadurch wird es möglich, die zusammengehörigigen MainFrames und TestFrames zu identifizieren.

9.3 Klasse CHOICEDATA

extends java.lang.Object

Die Klasse CHOICEDATA speichert die Angaben einer Auswahlliste, d. h. die Angabe für den Benutzer (UI-Wert) und den zugehörigen Wert in der Datenbank (DB-Wert). Diese Angaben können aus einem String ausgelesen werden, und in zwei Formaten ausgegeben werden.

Attribute:

-$EMPTY_UI: String

Der String, den der Benutzer im CHOICEEDITFRAME angezeigt bekommt, um die Position eines UI-Wertes anzugeben.

-$EMPTY_DB: String

Der String, den der Benutzer im ChoiceEditFrame angezeigt bekommt, um die Position eines DB-Wertes anzugeben.

-$EMPTY_LI: String

Der String, den der Benutzer im ChoiceEditFrame angezeigt bekommt, und der ihm eine Leerzeile vorschlägt.

Konstruktoren:

+ChoiceData (choiceString: String)

Erzeugt neue ChoiceData aus dem String, in dem abwechselnd UI- und DB-Werte stehen. Strings, die den EMPTY_XX-Konstanten entsprechen, werden ignoriert.

Methoden:

+addChoice (ui: String, db: String): void

Fügt die Kombination *ui*/*db* der Auswahlliste hinzu.

+clone(): Object

Gibt eine Kopie dieser ChoiceData zurück.

+$emptyString(): String

Gibt den String zurück, der bei einer leeren Auswahlliste angezeigt werden soll.

+isOK(): boolean

Gibt an, ob der eingebene String fehlerfrei geparst werden konnte.

+size(): int

Gibt die Anzahl der Alternativen in der Auswahlliste an.

+numberOfLines: int

Gibt die Anzahl der Zeilen in der String-Ausgabe an.

+width(): int

Gibt die maximale benötigte Breite für die Ausgabe der UI- und DB-Werte an.

+toString(): String

Gibt einen String zurück, in dem der Reihe nach die UI- und DB-Werte stehen, wobei jedes Paar durch eine Freizeile vom nächsten getrennt ist.

+toResourceBundle (prefix: String): String

Gibt die Auswahlliste in dem Format an, wie sie im ResourceBundle `Sprache.properties` gebraucht wird. Das Präfix *prefix* muss dabei den kompletten String enthalten, der vor `.ui=` bzw. `.db=` gestellt werden soll.

9.4 Klasse ChoiceEditFrame

extends Frame

implements ActionListener

Der ChoiceEditFrame (Verzeichnis `metamask/ui/`) bietet einen Frame, in dem eine Auswahlliste editiert werden kann.

Konstruktoren:

+ChoiceEditFrame (cel: ChoiceEditListener, parent: Frame, name: String, choices: ChoiceData)

Erzeugt einen neuen ChoiceEditFrame über dem Frame *parent*, der dem ChoiceEditListener *cel* das Beenden mitteilt. Der Frame hat den Namen *name* und zeigt die Auswahlliste *choices* an.

Methoden:

+actionPerformed (ActionEvent e): void

Implementation des ActionListener-Interfaces.

9.5 Interface ChoiceEditListener

Der ChoiceEditListener (Verzeichnis `metamask/ui/`) ist ein Interface zur Kommunikation eines ChoiceEditFrames mit einem Arbeitsablauf.

Methoden:

+choiceChangedText (newChoices: ChoiceData): void

Teilt dem Arbeitsablauf mit, dass der Text der Auswahl geändert wurde; der neue Inhalt der Auswahlliste steht in der übergebenen Klasse ChoiceData.

+choiceCancel(): void

Teilt dem Arbeitsablauf mit, dass der Text der Auswahl nicht geändert werden soll.

9.6 Klasse DataManager

extends java.lang.Object

implements Cloneable

Der DataManager (Verzeichnis `metamask/dok/`) ist die zentrale datenspeichernde Klasse von *MetaMask*. Er ersetzt dadurch den Zugriff auf die Datenbank. Hier sind sämtliche Informationen über das Projekt und die bearbeitete Maske enthalten, und können vom Arbeitsablauf ausgelesen oder in ein ResourceBundle geschrieben werden.

Attribute:

-$ProjectFullPath: String

Der voll qualifizierte Pfad des Projektes.

-$ProjectJavaPath: String

Der Pfad des Projektes, ausgehend vom Java-Wurzelverzeichnis.

-$ProjectName: String

Der Name des Projektes.

-$SkeletonFullPath: String

Der voll qualifizierte Pfad der Template-Files.

-$tableOwner: String

Der Datenbankbesitzer der Tabelle, die dem Projekt zugrundeliegt.

-$tableName: String

Der Name der Datenbanktabelle, die dem Projekt zugrundeliegt.

-$primaryKeys: Vector

Der Vektor mit den Primärschlüsseln der Tabelle, die dem Projekt zugrundeliegt.

-layout: Vector

Der Vektor, der die Zeilenvektoren der erzeugten Maske enthält.

-$rbNumber: int

Die Nummer des ResourceBundles, das für einen Testframe erzeugt wurde.

Konstruktoren:

+DataManager()

Erzeugt einen neuen, leeren DataManager.

Methoden:

+checkLayout (originals: boolean): void

Erzeugt ein neues Layout aus den als sichtbar markierten FieldEntries. Wenn der Parameter *originals* auf `True` gesetzt ist, werden die errechneten Positionen in den FieldEntries als „Originalpositionen“ eingetragen.

+cleanUpRBs(): void

Löscht die für Testframes erzeugten ResourceBundles.

+clearLayout(): void

Löscht das bisher erzeugte Layout. Die einzelnen Maskenelemente behalten ihre momentane Position allerdings bei.

+clone(): Object

Gibt eine Kopie dieses DataManagers zurück.

+contains (dbName: String): boolean

Gibt zurück, ob ein FieldEntry mit dem Namen *dbName* vorhanden ist.

+createDBNameVector(): Vector

Erzeugt einen sortierten Vektor der Namen sämtlicher FieldEntries und gibt diesen zurück.

+deleteFromLayout (fe: FieldEntry): void

Löscht das Feld im Layout, an dem sich der FieldEntry *fe* befindet.

+elements(): Enumeration

Gibt die FieldEntries als Aufzählung zurück; die Reihenfolge entspricht dabei der der Spalten in der Datenbankrelation.

+getChoices(dbName: String): ChoiceData

Holt die Auswahlliste für den FieldEntry mit dem Namen *dbName*. Die Angabe `Null` für *dbName* holt die Auswahlliste, die unabhängig von FieldEntries abgespeichert wurde.

+getFieldEntry (dbName: String): FieldEntry

Holt den FieldEntry mit dem Namen *dbName* oder gibt `Null` zurück, wenn kein solcher FieldEnty existiert.

+getFirstName(): String

Gibt den Namen des im Layout ersten FieldEntries zurück.

+getNr (dbName: String): int

Gibt die Nummer des FieldEntries mit dem Namen *dbName*.

+getNrOfRows(): int

Gibt die Anzahl der Zeilen in der erzeugten Maske zurück.

+getProjectName(): String

Gibt den Namen des Projektes zurück.

+getRB(): ResourceBundle

Holt das ResourceBundle des Projekts.

+getTestRBFullName(): String

Holt den voll qualifizierten Namen des ResourceBundles für das aktuelle Testfenster.

+getTestRBJavaName(): String

Holt den Namen des ResourceBundles für das aktuelle Testfenster im Java-Format.

+getUnnamedFieldEntry(): FieldEntry

Holt einen neuen FieldEntry, der noch keinen Namen besitzt und daher noch nicht in die normale Speicherung übernommen ist. Dies geschieht mittels der Methode `insertFieldEntry`.

+insertFieldEntry(): boolean

Setzt einen FieldEntry, der vorher mittels `getUnnamedFieldEntry` erzeugt wurde, in die normale Speicherung ein. Der FieldEntry muss mittlerweile einen Namen erhalten haben, sonst ist der Aufruf wirkungslos.

+isFirst (fe: FieldEntry): boolean

Gibt zurück, ob *fe* der erste FieldEntry im Layout ist.

+isLast (fe: FieldEntry): boolean

Gibt zurück, ob *fe* der letzte FieldEntry im Layout ist.

+isMetaMask(): boolean

Gibt an, ob es sich beim bearbeiteten Projekt um *MetaMask* selbst handelt.

+isModified(): boolean

Gibt zurück, ob ein FieldEntry des DataManagers modifiziert wurde.

+isNew(): boolean

Gibt zurück, ob das Projekt neu aus einer Datenbanktabelle erzeugt wurde.

+length(): int

Gibt die Anzahl der Maskenelemente zurück.

+load(): void

Lädt die entsprechenden Passagen aus den instantiierten Klassenfiles eines bereits existierenden Projektes und ruft die entsprechenden Initialisierungsmethoden auf.

+LoadDatabaseDescription (sqla: SQLAssistant): boolean

Lädt die Beschreibung der Datenbanktabelle aus dem SQLAssistant *sqla*. Danach werden die entsprechenden Initialisationen ausgeführt. Gibt den Wert `False` zurück, wenn ein Datenbankfehler aufgetreten ist.

+LoadFromFiles (SortArray: String, SetCheckers: String): void

Lädt die Namen der Datenbankfelder und eine grundlegende Beschreibung aus den beiden übergebenen Strings, die in dieser Form aus den instantiierten Klassenfiles gelesen werden sollten.

+moveDownFrom (fe: FieldEntry): FieldEntry

Gibt den im Layout dem FieldEntry *fe* nachfolgenden FieldEntry zurück, oder *fe*, falls dies der letzte sein sollte.

+moveFirst(): FieldEntry

Gibt den im Layout ersten FieldEntry zurück.

+moveLast(): FieldEntry

Gibt den im Layout letzten FieldEntry zurück.

+moveUpFrom (fe: FieldEntry): FieldEntry

Gibt den im Layout dem FieldEntry *fe* vorangehenden FieldEntry zurück, oder *fe*, falls dies der erste sein sollte.

+newChoiceFor (dbName: String, tccUI: String, tccDB: String): boolean

Fügt an die Auswahlliste des FieldEntries mit dem Namen *dbName* eine neue Auswahl an.

+packLayout(): void

Löscht eventuell vorhandene freie Felder oder Zeilen aus dem Layout.

+projectExistedBefore(): boolean

Gibt zurück, ob das Projekt aus bereits existierenden Klassenfiles gelesen wurde.

+readFlags (NewArray: String, SetCheckers: String, UpdateArray: String): void

Liest einige Flags für die Maskenelemente aus den übergebenen Strings aus, die in diesem Format aus den bereits instantiierten Klassenfiles gelesen werden sollten.

+readResourceBundle (rb: ResourceBundle): void

Liest die Daten, die im ResourceBundle *rb* enthalten sind, aus und setzt die FieldEntries entsprechend.

+removeFieldEntry (name: String): boolean

Entfernt einen FieldEntry aus der Speicherung. Dies ist nur für nachträglich erzeugte FieldEntries empfehlenswert, da sonst eine Datenbankspalte keine Entsprechung mehr besitzt. *MetaMask* fängt eine solche Fehleingabe des Benutzers im Arbeitsablauf ab.

+removeNewChoices(): void

Löscht eine Auswahlliste, die noch zu keinem FieldEntry hinzugefügt wurde.

+reposition (fe: FieldEntry, toRow: int, toCol: int): void

Setzt den FieldEntry *fe* auf die Position *toCol* in der Zeile *toRow*. Ein Wert von 0 für *toRow* löscht den FieldEntry aus dem Layout; ein Wert von 0 für *toCol* setzt den FieldEntry ans Ende der Zeile *toRow* und ein Wert von -1 für *toCol* fügt den FieldEntry als neue *toRow*te Zeile ein.

+save (for_real: boolean): void

Speichert die eingegebenen Daten in entsprechenden Klassenfiles. Wenn das Projekt neu aus einer Datenbank erzeugt wurde, werden sämtliche Klassen neu aus den Templates instantiiert, ansonsten werden die Änderungen an die entsprechenden Stellen der bereits existierenden Klassenfiles eingetragen. Außerdem werden die ResouceBundles neu geschrieben.

Ist *for_real* auf `False` gesetzt, d. h. es soll nur ein ResouceBundle für ein Testfenster geschrieben werden, so werden die Klassenfiles weder instantiiert noch modifiziert.

+$setBasic (ProjectFullPath: String, ProjectJavaPath: String, ProjectName: String, tableOwner: String, tableName: String): void

Setzt die entsprechenden Werte, die für alle DataManager gleich bleiben.

+setChoices (dbName: String, newChoices: ChoiceData): void

Setzt die Auswahlliste für den FieldEntry mit dem Namen *dbName* auf *newChoices*. Eine eventuell vorher vorhandene Auswahlliste wird gelöscht. Die Angabe `Null` für *dbName* speichert die Auswahlliste unabhängig von einem FieldEntry; es kann aber auch hier nur eine gespeichert werden.

+$setProjectExistedBefore (peb: boolean): void

Teilt dem DataManager mit, dass es sich um ein Projekt handelt, was schon existierte, d. h. nicht aus der Datenbank ausgelesen werden soll.

+$setSkeleton (SkeletonFullPath: String): void

Setzt den voll qualifizierten Pfad, an dem die Templatefiles gefunden werden können.

+writeAllArray(): String

Gibt die Namen der Spalten in der Datenbankrelation in dem Format zurück, wie es im DokumentDatensatz und im SQLAssistant benötigt wird.

-writeBlocked(): String

Gibt die Namen der FieldEntries, die in bestimmten Programmzuständen deaktiviert sein sollen, in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeCheckers(): String

Gibt die Aufrufe der entsprechenden FieldChecker für die Eingabefelder in dem Format zurück, wie es im DokumentDatensatz benötigt wird.

-writeChoices(): String

Gibt die Auswahllisten in dem Format zurück, wie es im ResouceBundle `Sprache.properties` benötigt wird.

+writeConst(): String

Gibt die Namen der ResourceBundles in dem Format zurück, wie es im Arbeitsablauf benötigt wird.

+writeCursorDown(): String

Gibt die Zielfelder für die Abwärtsbewegung in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeCursorUp(): String

Gibt die Zielfelder für die Aufwärtsbewegung in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeDefList(): String

Gibt die Namen der Spalten in der Datenbankrelation in dem Format zurück, wie es im DokumentDatensatz und im SQLAssistant benötigt wird.

+writeFocushelp(): String

Gibt die Focushelp-Strings in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeLayout(): String

Gibt das Layout der Maske in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeListCommands(): String

Gibt die Befehlszeilen für die Knöpfe der TextColFieldwithList in dem Format zurück, wie es im Arbeitsablauf benötigt wird.

+writeNames(): String

Gibt „tableOwner“ und „tableName“ in dem Format zurück, wie es im SQLAssistant benötigt wird.

+writeNewArray(): String

Gibt die Namen der Spalten in der Datenbankrelation für einen neuen Datensatz in dem Format zurück, wie es im SQLAssistant benötigt wird.

+writeOrderBy(): String

Gibt die Formulierung des OrderBy-Menüs in dem Format zurück, wie es im ResourceBundle `Sprache.properties` benötigt wird.

+writeParsers(): String

Gibt die Aufrufe der entsprechenden Parser für die Eingabefelder in dem Format zurück, wie es in den Kriterien benötigt wird.

+writePermission(): String

Gibt die Formulierung der SQL-Anfrage nach der Schreiberlaubnis in dem Format zurück, wie es im SQLAssistant benötigt wird.

+writePrimaryKeys(): String

Gibt die SQL-Anfrage nach den Primärschlüsseln in dem Format zurück, wie es im DokumentDatensatz benötigt wird.

+writeSortArray(): String

Gibt die Namen der Spalten in der Datenbankrelation in der Reihenfolge der Datenbankspalten in dem Format zurück, wie es im SQLAssistant benötigt wird.

+writeUpdateArray(): String

Gibt die Namen der Spalten in der Datenbankrelation, die modifiziert werden können, in dem Format zurück, wie es im SQLAssistant benötigt wird.

-writeVersion(): String

Gibt einen Versions-String mit dem aktuellen Datum für das Projekt zurück.

-initFieldEntry (dbName: String, dbType: int, dbLength: int, isNullable: boolean, dataType: int): void

Initialisiert den FieldEntry mit dem Namen *dbName* mit den entsprechenden Werten.

-initFieldEntry (fe: FieldEntry, dbType: int, dbLength: int, isNullable: boolean, dataType: int): void

Initialisiert den FieldEntry *fe* mit den entsprechenden Werten.

-moveUpFrom (row: int, col: int): FieldEntry

Gibt den letzten FieldEntry vor der Position *col* der Zeile *row* zurück, oder `Null`, falls es keinen solchen gibt.

-moveDownFrom (row: int, col: int): FieldEntry

Gibt den ersten FieldEntry nach der Position *col* der Zeile *row* zurück, oder `Null`, falls es keinen solchen gibt.

-newFieldEntry (dbName: String): FieldEntry

Erzeugt einen neuen FieldEntry mit dem Namen *dbName* und gibt ihn zurück, nachdem er in die Speicherung eingefügt wurde.

9.7 Klasse FieldEntry

extends java.lang.Object

implements Cloneable

Die Klasse FieldEntry (Verzeichnis `metamask/dok/`) entspricht einem Maskenelement der erzeugten Maske. Hier werden sämtliche Informationen, wie Position innerhalb der Maske, Größe etc. sowie deren Modifikationen gespeichert.

Attribute:

+$IntegerVals: Vector

Listet diejenigen Eigenschaften auf, die Integerwerte enthalten.

+$keys: String[]

Ein Array mit den Namen aller Eigenschaften.

-orig: Hashtable

Die Originalwerte der einzelnen Eigenschaften.

-modi: Hashtable

Die Modifikationen an den einzelnen Eigenschaften.

-deleted: boolean

Gibt an, ob dieser FieldEntry gelöscht werden soll.

Konstruktoren:

+FieldEntry()

Erzeugt einen neuen FieldEntry, der sämtliche Eigenschaften auf Defaultwerte gesetzt bekommt.

Methoden:

+acceptAsOriginal(): void

Überträgt sämtliche Modifikationen in die Originalwerte; der FieldEntry ist damit wieder unmodifiziert.

+clone(): void

Gibt eine Kopie dieses FieldEntries zurück.

+get (name: String): String

Gibt den aktuellen Wert der Eigenschaft *name* als String zurück; diese Methode funktioniert auch dann, wenn diese Eigenschaft Boolean- oder Integerwerte erwartet.

+getBoolean (name: String): boolean

Holt den aktuellen Wert der Eigenschaft *name*, die Boolean-Werte akzeptiert.

+getInt (name: String): int

Holt den aktuellen Wert der Eigenschaft *name*, die Integer-Werte akzeptiert.

+getOrig (name: String): String

Gibt den Originalwert der Eigenschaft *name* als String zurück; diese Methode funktioniert auch dann, wenn diese Eigenschaft Boolean- oder Integerwerte erwartet.

+getOrigBoolean (name: String): boolean

Holt den Originalwert der Eigenschaft *name*, die Boolean-Werte akzeptiert.

+isDeleted(): boolean

Gibt zurück, ob dieser FieldEntry als gelöscht markiert ist.

+isModified(): boolean

Gibt zurück, ob eine Eigenschaft dieses FieldEntries modifiziert wurde.

+isModified (name: String): boolean

Gibt zurück, ob die Eigenschaft *name* in diesem FieldEntry modifiziert wurde.

+restore(): void

Löscht sämtliche Modifikationen; alle Eigenschaften sind damit wieder auf ihre Originalwerte gesetzt.

+restore (name: String): void

Löscht eventuelle Modifikationen der Eigenschaft *name*; sie ist damit wieder auf ihren Originalwert gesetzt.

+set (name: String, modiValue: boolean): boolean

Setzt den Wert *modiValue* als neuen Wert für die Eigenschaft *name*, die Booleanwerte akzeptiert.

+set (name: String, modiValue: int): boolean

Setzt den Wert *modiValue* als neuen Wert für die Eigenschaft *name*, die Integer-Werte akzeptiert.

+set (name: String, modiValue: String): boolean

Setzt den Wert *modiValue* als neuen Wert für die Eigenschaft *name*, die String-Werte akzeptiert.

+setDeleted (deleted: boolean): void

Markiert diesen FIELDENTRY als gelöscht oder nicht gelöscht je nach Wert von *deleted*.

+setOrig (name: String, origValue: boolean): boolean

Setzt den Wert *origValue* als neuen Originalwert für die Eigentschaft *name*, die Boolean-Werte akzeptiert.

+setOrig (name: String, origValue: int): boolean

Setzt den Wert *origValue* als neuen Originalwert für die Eigentschaft *name*, die Integer-Werte akzeptiert.

+setOrig (name: String, origValue: String): boolean

Setzt den Wert *origValue* als neuen Originalwert für die Eigentschaft *name*, die String-Werte akzeptiert.

-insertCharacteristic (name: String, origValue: boolean): void

Fügt eine neue Eigenschaft mit dem Namen *name* ein, die Boolean-Werte akzeptiert und auf *origValue* gesetzt ist.

-insertCharacteristic (name: String, origValue: int): void

Fügt eine neue Eigenschaft mit dem Namen *name* ein, die Integer-Werte akzeptiert und auf *origValue* gesetzt ist.

-insertCharacteristic (name: String, origValue: String): void

Fügt eine neue Eigenschaft mit dem Namen *name* ein, die String-Werte akzeptiert und auf *origValue* gesetzt ist.

9.8 Klasse REPOSITIONFRAME

extends java.awt.Frame

implements ActionListener, TextColListener

Der REPOSITIONFRAME (Verzeichnis `metamask/ui/`) ist das Fenster, in dem der Benutzer die neue Position eines Maskenelements eingeben kann.

Konstruktoren:

+RepositionFrame (rfl: RepositionFrameListener, parent: Frame, name: String, initial_row: int, initial_col: int)

Erzeugt einen neuen RepositionFrame, der über dem Frame *parent* erscheint und dem RepositionFrameListener *rfl* das Beenden mitteilt. Das Maskenelement, das umpositioniert werden soll, trägt den Namen *name* und befindet sich ursprünglich in der Zeile *initial_row* und der Position *initial_col*.

Methoden:

+actionPerformed (ActionEvent e): void

Implementation des ActionListener-Interfaces.

+pleaseCenterOver (parent: Frame): void

Zentriert den Frame über dem Frame *parent*.

+textColValueChanged (tci: TextColItem, name: String, text: String): void

Implementation des TextColListener-Interfaces.

9.9 Interface RepositionFrameListener

Der RepositionFrameListener (Verzeichnis `metamask/ui/`) ist ein Interface zur Kommunikation zwischen einen RepositionFrame und einem Arbeitsablauf.

Methoden:

+executeRepositionCommand (ac: String, row: int, col: int, ok: boolean): void

Wird aufgerufen, wenn im RepositionFrame der Befehl *ac* ausgeführt werden soll. Die Angaben *row* und *col* geben Zeile und Position innerhalb der Zeile an, die der Benutzer eingetragen hat. Der Parameter *ok* gibt an, ob die neue Position angenommen werden soll.

9.10 Klasse SQLAssistant

extends java.lang.Object

Der SQLAssistant (Verzeichnis `metamask/dok/`) hat mit dem SQL-Assistenten der automatisch erzeugen Masken außer der logischen Position nicht viel gemeinsam. Das liegt daran, dass *MetaMask* nicht ständig auf die Datenbank zugreifen muss, und bei dem einen notwendigen Zugriff nicht die Daten aus der Tabelle auslesen soll, sondern deren Beschreibung.

Konstruktoren:

+SQLAssistant (con: Connection)

Erzeugt einen neuen SQLAssistant auf der Connection *con*.

Methoden:

+getDescOf (tableOwner: String, tableName: String): ResultSetMetaData

Holt die Beschreibung der Tabelle *tableName* des Datenbankbenutzers *tableOwner*. Dies geschieht durch das Holen einer leeren Anfrage (`SELECT * FROM` user.table `WHERE 1=0`), deren ResulSetMetaData dann zurückgegeben wird.

+getPrimaryKeysOf (tableOwner: String, tableName: String): Vector

Holt die Primärschlüssel der Tabelle *tableName* des Datenbankbenutzers *tableOwner*. Diese werden als Vektor zurückgegeben.

9.11 Klasse AblaufVerwalter (Erweitert)

Der AblaufVerwalter wurde nicht viel modifiziert; allerdings wird ein Exemplar des unmodifizierten DataManagers gespeichert, von dem jedem neu geöffneten Arbeitsablauf eine Kopie übergeben wird. So ist es möglich, in verschiedenen Arbeitsabläufen unterschiedliche Darstellungsalternativen vom selben Ausgangspunkt aus zu vergleichen.

Attribute:

-$original_dm: DataManager

Der DataManager, wie er beim Start von *MetaMask* eingelesen wurde, bzw. wie er in einem Arbeitsablauf vom Benutzer als neue Maske übernommen wurde.

Methoden:

+$setDM (dm: DataManager): void

Setzt den DataManager, dessen Kopie jedem neu geöffneten Arbeitsablauf mitgegeben wird, auf *dm*.

9.12 Klasse FlitLoginFrame (Erweitert)

Der FlitLoginFrame muss mehr Eingabe akzeptieren als diejenigen der automatisch instantiierten Masken, dementsprechend enthält er mehr graphische Elemente und erweiterte Methoden. Da diese sich jedoch in ihrer generellen Beschreibung nicht von denen des üblichen FlitLoginFrame unterscheiden, sei hier auf dessen Beschreibung in Abschnitt 7.33 verwiesen.

Kapitel 10

Besonderheiten in der Implementation von *Litera*

10.1 die speziellen Anforderungen

Die Maske *Litera* ist für die Literaturdatenbank entworfen. Für diese Relation soll es möglich sein, Dokumente zu bestellen. Diese Bestellungen werden darüber hinaus in einer separaten Kontrollliste protokolliert, aus der sie wieder gelöscht werden, wenn die Bücher angekommen sind und in die Hauptrelation aufgenommen werden.

Dokumente können auf zwei unterschiedliche Arten bestellt werden: Man kann ein Dokument, das sich in einem Suchergebnis befindet und momentan auf dem Bildschirm angezeigt wird, den Eintrag im Feld „Status“ auf „zu bestellen“ ändern. Dadurch wird das Dokument in der Datenbank entsprechend markiert, und bei der nächsten Abfrage nach Dokumenten, die bestellt werden sollen, mit ausgegeben.

Andererseits kann ein Benutzer auch Dokumente in einen „Warenkorb“ sammeln. Dies wird nicht direkt in der Datenbank mitprotokolliert; die Dokumente sind also nur so lange markiert, bis der Benutzer das Programm *Litera* wieder beendet. Wird die Einzelbestellung des Warenkorbes allerdings ausgeführt, so wird dies selbstverständlich in die Datenbank mit aufgenommen.

Die für diese Instantiierung der Standardmaske hinzugekommenen Klassen, die in den folgenden Abschitten dokumentiert werden, spiegeln die neuen Anforderungen im Zusammenhang mit den Bestellungen wieder. Außerdem mussten Änderungen an den automatisch instantiierten Klassen vorgenommen werden, die in Abschnitt 10.8 und den folgenden dokumentiert sind.

Durch die Tatsache, dass in der Hauptrelation die Dokumente aller drei Abteilungen des Instituts für Informatik zusammengeführt sind, ist ein erweitertes Rollensystem in der Oracle-Datenbank nötig geworden, das in Abschnitt 10.11 beschrieben wird.

10.2 Klasse BestellFrame

extends java.awt.Frame

implements KeyListener, WindowListener, ActionListener

Der BestellFrame (Verzeichnis `litera/ui/`) ist ein Fenster von Litera, das eine Auswahlliste enthält, die Lieferanten und die zugehörige Bestellsumme angibt. Der Inhalt der Liste wird durch einen Vektor angegeben, der abwechselnd Strings (Lieferantenname) und FloatZahlen (Bestellsumme) enthält.

Konstruktoren:

+BestellFrame (v: Vector, bfl: BestellFrameListener, name: String, font: Font, RechnungsWaehrungUI: String)

Erzeugt einen neuen BestellFrame aus dem im Vektor *v* gegebenen Liste, die abwechselnd Strings (Lieferanten) und FloatZahlen (Bestellsumme) enthält. Es muss ein BestellFrameListener *bfl* angegeben werden, der über die auftretenden Events benachrichtigt wird. Der Name *name* dient ausschließlich dem BestellFrameListener zur Identifikation. Die Angabe des Strings *RechnungsWaehrungUI* ist lediglich zur Bezeichung der Zahlen in der Liste nötig.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+displayAndToFront(): void

Macht diesen BestellFrame sichtbar und bringt ihn nach vorne.

+getPreferredSize(): Dimension

Holt die bevorzugte Größe des BestellFrames.

+keyPressed (e: KeyEvent): void

Implementation des KeyListener-Interfaces.

+keyReleased (e: KeyEvent): void

Implementation des KeyListener-Interfaces; leer.

+keyTyped (e: KeyEvent): void

Implementation des KeyListener-Interfaces; leer.

+pleaseBeAsWideAs (w: int): void

Schlägt dem BestellFrame eine gewünschte Breite vor.

+pleaseCenterOver (parent: Frame): void

Positioniert den BestellFrame so auf dem Bildschirm, dass er zentriert über dem Frame *parent* erscheint, ohne sich jedoch über den Bildschirmrand hinaus zu erstrecken.

+pleaseDispose(): void

Veranlasst den BestellFrame, sich zu schließen und alle Resourcen freizugeben. Darf mehrfach aufgerufen werden – alle Aufrufe nach dem ersten sind dann wirkungslos.

+pleasePositionHere (p: Point): void

Positioniert den BestellFrame auf dem Bildschirm. Die Position p wird dabei nur als Empfehlung betrachtet, von der abgewichen wird, falls der BestellFrame sonst nicht mehr vollständig auf dem Bildschirm sichtbar wäre.

+requestFocus(): void

Veranlasst den BestellFrame, den Eingabe-Focus in die Liste zu holen.

+windowActivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowClosed (e: WindowEvent): void

Implementation des WindowListnener-Interfaces.

+windowClosing (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowDeactivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowDeiconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowIconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; schließt den BestellFrame.

+windowOpened (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

10.3 Interface BestellFrameListener

Der BestellFrameListener (Verzeichnis `litera/ui/`) stellt ein Interface zur Kommunikation zwischen einem BestellFrame und einem Arbeitsablauf dar.

Methoden:

+executeBestellCommand (ac: String, value: String): void

Wird aufgerufen, wenn der Befehl *ac* in einem BestellFrame ausgeführt werden soll. Der Wert, den der Benutzer ausgewählt hat, wird in *value* übergeben.

10.4 Klasse BestellListendarstFrame

extends java.awt.Frame

implements WindowListener, ActionListener

Der BestellListendarstFrame (Verzeichnis `litera/ui/`) ist ein Fenster, das eine vorbereitete Bestellliste anzeigt.

Konstruktoren:

+BestellListdarstFrame (s: String, font: Font, bldl: BestellListendarstListener)

Erzeugt einen neuen BestellListendarstFrame, der den String *s* anzeigt. Der übergebene BestellListendarstListener *bldl* wird auf die auftretenden Events reagieren.

Methoden:

+actionPerformed (e: ActionEvent): void

Implementation des ActionListener-Interfaces.

+windowActivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowClosed (e: WindowEvent): void

Implementation des WindowListnener-Interfaces.

+windowClosing (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

+windowDeactivated (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowDeiconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowIconified (e: WindowEvent): void

Implementation des WindowListener-Interfaces; leer.

+windowOpened (e: WindowEvent): void

Implementation des WindowListener-Interfaces.

10.5 Interface BestellListdarstListener

Der BestellListdarstListener (Verzeichnis `litera/ui/`) ist ein Interface zur Kommunikation zwischen einem BestellListdarstFrame und einem Arbeitsablauf.

Methoden:

+executeBestellListdarstCommand (do_best: boolean): void

Wird aufgerufen, wenn der BestellListdarstFrame geschlossen wird. Der Benutzer hat ausgewählt, ob die Bestellung durchgeführt werden soll (und in die Kontrolllisten eingetragen), wenn *do_best* `True` ist, ansonsten will der Benutzer die Bestellung abbrechen.

10.6 Klasse KontrollListManager

extends ListManager

Der KontrollListManager (Verzeichnis `litera/dok/`) kann aus einem SuchErgebnis eine Übersicht in Form eines Strings erstellen. Der String enthält in jeder Zeile einen Datensatz des SuchErgebnisses. Start- und Endposition innerhalb des SuchErgebnisses sind frei wählbar. Ebenso ist frei wählbar, welche Spalten die Liste in welcher Reihenfolge enthalten soll.

Die Erweiterung gegenüber des ListManagers besteht darin, dass eine zusätzliche Zeile ausgegeben wird, wenn eine neuer Lieferant angegeben wird. Zur Einstellung des Formats sei auf die Klasse ListManager verwiesen.

Konstruktoren:

+KontrollListManager (se: SuchErgebnis, spalten: String)

Erzeugt einen neuen KontrollListManager, der auf dem SuchErgebnis *se* arbeitet und sein Format aus dem String *spalten* ausliest.

Methoden:

+getList (separateByLine: boolean): String

Erzeugt den String, der die Ausgabe im eingestellten Format enthält. Wenn *separateByLine* `True` ist, werden zwei Zeilen jeweils durch eine Leerzeile getrennt.

10.7 Klasse Warenkorb

extends java.lang.Object

Der Warenkorb ermöglicht die Sammlung von DokumentDatensätze zur Einzelbestellung.

Konstruktoren:

+Warenkorb()

Default-Konstruktor; leer.

Methoden:

+clear(): void

Löscht alle Einträge aus dem WARENKORB.

+delete (dd: DOKUMENTDATENSATZ): boolean

Löscht den DOKUMENTDATENSATZ *dd* aus dem WARENKORB. Gibt `False` zurück, falls der Datensatz gar nicht im WARENKORB vorhanden war, sonst `True`. Auf jeden Fall ist der Datensatz danach nicht (mehr) im WARENKORB.

+exists (dd: DOKUMENTDATENSATZ): int

Prüft, ob der DOKUMENTDATENSATZ *dd* sich bereits im WARENKORB befindet, oder schon als bestellt markiert ist und daher nicht in den WARENKORB gelegt werden können soll. Der Rückgabewert ist 1, falls er sich schon im WARENKORB befindet, 2, falls er als bestellt markiert ist, und 0 sonst.

+insert (dd: DOKUMENTDATENSATZ): boolean

Fügt den DOKUMENTDATENSATZ *dd* in den WARENKORB ein. Gibt `False` zurück, falls der Datensatz schon vorhanden war, sonst `True`. Auf jeden Fall ist der Datensatz danach im WARENKORB.

+isEmpty(): boolean

Gibt an, ob der WARENKORB leer ist.

+locate (dd: DOKUMENTDATENSATZ): int

Prüft, ob sich der DOKUMENTDATENSATZ *dd* bereits im WARENKORB befindet, und gibt gegebenenfalls dessen Position zurück. Ist er nicht vorhanden, wird -1 zurückgegeben.

+toString(): String

Erzeugt einen String, der die Einträge im WARENKORB in der Form ('doknr','abteilung') enthält, um sie in einer SQL-Anfrage auswerten zu können.

10.8 Klasse ARBEITSABLAUF (Erweitert)

Der ARBEITSABLAUF wurde dahingehend erweitert, dass er die Abteilung des Benutzers gesondert behandelt. Außerdem wurden die Funkionalitäten für die Einzelbestellungen (WARENKORB, Menübedienung) hinzugefügt, sowie die Implementationen der entsprechenden Listener.

Attribute:

-einzelbestellung: WARENKORB

Der WARENKORB, in dem die Einzelbestellung zusammengestellt wird.

-$default_abteilung: String

Benutzerabhängige Default-Einstellung für die Abteilung. Bei schreibberechtigen Benutzern ist dies die einzige Abteilung, der sie Dokumente hinzufügen können.

-$ABTEILUNG_EGAL: String

Eine String-Konstante, die angibt, dass der Wert der Abteilung irrelevant ist.

Methoden:

+$setDefaultAbteilung (def_abteilung: String): void

Setzt den Defaultwert der Abteilung für den Benutzer.

+executeBestellCommand (ac: String, value: String): void

Implementation des BestellFrameListener-Interfaces.

+executeBestellListdarstCommand (do_best: boolean): void

Implementation des BestellListendarstListener-Interfaces. Übernimmt die Bestellung in die Literatur-Relation und die Kontrolltabelle.

10.9 Klasse SQLAssistant (Erweitert)

Der SQLAssistant wurde um die Funktionalität erweitert, die Default-Abteilung für einen Benutzer aus der Datenbank auslesen zu können, und die Verwaltung der Kontrollliste zu übernehmen.

Methoden:

+getDefaultAbteilung (name: String): String

Liefert einen String mit dem Defaultwert für die Abteilung des Benutzers mit dem LoginNamen *name*. Kann auch „Egal“ zurückgeben.

+getBestDatum (doknr: String, abteilung: String): String

Liest das Bestelldatum des durch *doknr* und *abteilung* eindeutig identifizierten Dokuments aus der Kontrolltabelle. Ist dieses Dokument nicht in die Kontrolltabelle eingetragen, so wird der String „nicht vorhanden“ zurückgegeben. Tritt beim Zugriff auf die Datenbank eine Exception auf, so wird der String „SQL-Zugriffsfehler“ zurückgegeben.

+deleteFromKontrolle(): void

Löscht diejenigen Einträge aus der Kontrollliste, bei denen der zugehörige Eintrag in der Haupttabelle nicht mehr als „bestellt“ markiert ist.

+insertIntoKontrolle (doknr: String, abteilung: String): void

Fügt ein Dokument, das über *doknr* und *abteilung* eindeutig identifiziert wird, in die Kontrollliste ein; dies protokolliert die Bestellungen.

10.10 ResourceBundle `Sprache.properties` (Erweitert)

Im ResourceBundle `Sprache.properties` wurde den erweiterten Anforderungen dadurch Rechnung getragen, dass – abgesehen von den Einträgen für die Auswahlliste der Währung und dem neuen Menü für die Einzelbestellungen und der Rechnungswährung – die Formatstrings `best.format` und `kontroll.format` hinzukamen, die das Listen-Ausgabeformat für die Bestellliste bzw. die Kontrollliste angeben.

10.11 Das erweiterte Rollensystem

Litera sieht vor, dass sich verschiedene Benutzer in die Datenbank einloggen können, wobei die Unterteilung feiner sein soll als „mit“ oder „ohne Schreibrecht“. Es ist vielmehr so, dass die meisten Benutzer zu einer der drei Informatik-Abteilungen gehören, und wenn sie Schreibrechte besitzen, auch nur Dokumente hinzufügen können, die zu dieser Abteilung gehören.

Dies wird realisiert durch eine weitere Datenbankrelation namens `default_ifi`, in der zu jedem Benutzer der entsprechende Buchstabe ('A', 'B' oder 'C') bzw. der Wert 'Egal' eingetragen ist. Ist ein Buchstabe eingetragen, so können nur Bücher der genannten Abteilung hinzugefügt werden; außerdem ist bei Suchanfragen das Eingabefeld „Abteilung“ zu Beginn auf diesen Wert gesetzt, kann aber verändert werden. Der Eintrag 'Egal' bewirkt, dass Dokumente aller Abteilungen hinzugefügt werden können.

Die Funktion dieser Datenbankrelation erfordert, dass sämtliche Benutzer der Maske *Litera* das Leserecht auf ihr erhalten. Hat ein Benutzer kein Leserecht auf dieser Relation (oder wird sein Name nicht gefunden), so erhält er auch kein Schreibrecht auf der Hauptrelation. Dies ist notwendig für den Fall des reinen Gast-Logins.

Darüber hinaus speichert *Litera* das Bestelldatum der Dokumente in der Kontrolltabelle `kontrolle`. Auf diese Relation müssen nur diejenigen Benutzer zugreifen können, die Bestellungen ausführen sollen, und das heißt, dass sie bereits Schreibrechte auf der Hauptrelation besitzen.

Abschließend sei hier noch die Einrichtung der notwendigen Benutzerrollen angegeben, wobei wie im allgemeinen Fall `litera_guest` für diejenigen Benutzer gedacht ist, die nur Leserechte auf der Hauptrelation besitzen (egal, ob sie einen Eintrag in der Relation `default_ifi` haben oder nicht), `litera_user` für diejenigen Benutzer, die Schreibrechte besitzen, und `litera_admin` für den Administrator.

```
create role litera_guest;
    grant select on literfassung to litera_guest;
    grant select on default_ifi to litera_guest;
```

```
create role litera_user;
    grant litera_guest to litera_user;
    grant insert, update, delete on literfassung to litera_user;
    grant select on litera_readwrite to litera_user;
    grant select, insert, update, delete on kontrolle to litera_user;


create role litera_admin;
    grant litera_user to litera_admin;
    grant insert, update, delete on default_ifi to litera_admin;
```

Der Administrator hat durch den Schreibzugriff auf die Relation `default_ifi` die Möglichkeit, neue Benutzer in diese Relation einzufügen oder deren Einträge zu verändern, um ihnen ggf. uneingeschränkten Zugriff auf die Hauptrelation zu gewähren.